The Dissertation Committee for Pak Ho Chung
certifies that this is the approved version of the following dissertation:

Collaborative Intrusion Prevention

Committee:

Aloysius K. Mok, Supervisor

Vitaly Shmatikov

Yin Zhang

Wenke Lee

David Gilliam

Collaborative Intrusion Prevention

by

Pak Ho Chung, B.Eng., M.S

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2009

Dedicated to those who will read this work...

Acknowledgments

I would like to thank the following people here: My parents, without them, this work will not be started, not to mention finished, for many obvious reasons; My supervisor, Prof Al. Mok, for the unfailing support and encouragement, as well as the "dare-try" attitude; Prof Vitaly Shmatikov, Prof Yin Zhang, Prof Wenke Lee and Dr David Gilliam, for serving on my dissertation committee; Dr Pan Aimin in MSRA, for all our discussions and his belief in my abilities/value; Mr Hsieh Shin Ru and Ms Vivian Lee, the pain and injuries they caused define me, as James T. Kirk said "They are things we carry with us, the things that make us who we are"; Amber Sun, for encouraging me to do my best in this dissertation. I am also very grateful to Vivian for the great, albeit short time we had together. I would also like to thank the following friends, each of them, at some point in time, helped maintain my mind at a not too unhealthy state, hear me whine, etc: C. Y. Koo, Raymond Lau, Li Yi, Yiu-Fai Sit, John Wong, Chappy Kwan, Simon Lam (not the UTCS faculty), Thomas Lo, Jeffery Chau, Derek Lee, Zhou Xia, Zhou Bo, Wu Zhong, Ving Lei, Su Xiaoxia, Nalini Belaramani, Liu Huiya (and the many more whose name I forget to mention here. forgive me).

Collaborative Intrusion Prevention

Publication No. _____

Pak Ho Chung, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Aloysius K. Mok

Intrusion Prevention Systems (IPSs) have long been proposed as a defense against attacks that propagate too fast for any manual response to be useful. While purely-network-based IPSs have the advantage of being easy to install and manage, research have shown that this class of systems are vulnerable to evasion [70, 65], and can be tricked into filtering normal traffic and create more harm than good [12, 13]. Based on these researches, we believe information about how the attacked hosts process the malicious input is essential to an effective and reliable IPS. In existing IPSs, honeypots are usually used to collect such information. The collected information will then be analyzed to generate countermeasures against the observed attack. Unfortunately, techniques that allow the honeypots in a network to be identified ([5, 71]) can render these IPSs useless. In particular, attacks can be designed to avoid targeting the identified honeypots. As a result, the IPSs will have no information about the attacks, and thus no countermeasure will ever be generated. The use of honeypots is also creating other practical issues which

limit the usefulness/feasibility of many host-based IPSs. We propose to solve these problems by duplicating the detection and analysis capability on every protected system; i.e., turning every host into a honeypot.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

When building provably secure system is beyond the limits of current technology [6] and considered economically infeasible by some [17], software vulnerabilities with serious security consequences will remain a fact of life in the near future. Currently, when a new vulnerability (security related or not) is discovered, the vendor of the software involved will manually generate patches for the vulnerability (usually in the form of fixed version of the faulty libraries), where owners/administrators of vulnerable systems will download and apply. However, it is well recognized that this manual generation and application of patches are too slow to protect vulnerable systems from attacks, with the most cited demonstration of this fact being the outbreak of the SQLSlammer worm (which, according to [60], infected more than 90% of all vulnerable hosts within 10 mins). Furthermore, as argued in [90], reliability issues also make patching an ineffective solution to problems with vulnerable software; many are reluctant to apply patches, for fear that the patching will disrupt the normal operation of their systems.

In response to these shortcomings of the current patching practice, many have proposed intrusion prevention systems (IPSs) as an alternative

solution. Simply put, IPSs are systems the detect attacks against a protected system/network, collect information about the attack, analyze the collected information and output "countermeasures" to stop future instances of the observed attack. Existing IPSs can be categorized into network-based and host-based. In this dissertation, we will focus on host-based IPSs, where we consider any system that analyzes how a vulnerable host processes attack traffic a host-based IPS, even though they may generate countermeasures for filtering network traffic. Example of host-based IPSs include [15, 63, 7, 77, 97, 54].

We choose to focus on host-based IPSs because analysis in [70, 65] seems to suggest that a purely network-based approach is inevitably vulnerable to polymorphic attacks. Furthermore, our research [12, 13] show that in what we called the allergy attack, many of these network-based IPSs can be induced into outputting attack signatures that will filter out a large portion of normal traffic, and effectively create a denial of service (DoS) against the protected network. All these research on attacks against network-based IPSs make us believe that an IPS must rely on some feedback from the protected systems.

In many existing host-based IPSs, attack information needed for generating countermeasures is collected on dedicated hosts, since the components for collecting such information is too heavy for production systems (we will elaborate on this point in Sect.3.1). Though they maybe labeled differently, these dedicated systems are effectively honeypots; systems with no value other than being attacked and compromised. However, we find that there are a few

drawbacks in using honeypots to collect information about attacks:

1. As shown in [5, 71], the addresses of honeypots can be known to the attackers, allowing them to avoid interaction with the honeypots. As a result, the IPSs will be unable to collect any information about the new attacks, and thus no countermeasures will be generated. Furthermore, the heavy-weighted analysis commonly performed in the honeypots also makes them easy target of DoS attacks. In short, the honeypots have become a single point of failure in the IPSs that use them to collect information about attacks; any failure in the honeypots will render the IPSs blind to new attacks, and leave the systems protected by the IPSs at the mercy of the attackers.

2. The great variety of OSs/applications, along with the many different versions of the same software running on different protected hosts is also creating some practical difficulties; a large number of honeypots may be needed so that attacks against hosts running rare OSs/applications, or a specific version of some software module will be covered. This can amount to a non-trivial cost to set up and manage all these honeypots (especially if one has to acquire all the different OSs/applications to be protected). We note that though targeting rare OSs/applications or specific version of a software module will significantly decrease the size of vulnerable population, these "specific attacks" still pose a significant threat. In fact, SQLSlammer, one of the most notorious worms in history,

can be considered an example of such attacks. First of all, SQLSlammer targets systems running Microsoft SQL server 2000 or Microsoft Desktop Engine 2000, which are far less common than applications like svchost.exe or lsass.exe (which are targeted by MSBlast and Sasser respectively). Furthermore, as argued in [22], the worm may work only on systems running specific versions of dynamic link libraries (DLLs), due to its use of static library address (though, it's not determined what versions of DLL are affected). Yet, SQLSlammer is proved more damaging than better designed, more portable worms that target common applications.

3. The passive nature of honeypots also makes them unsuitable for studying attacks that involve any user action (e.g. a successful attack on web browsers usually requires the victim to visit some contaminated web pages). Even though this problem may be alleviated by techniques like [92], we believe this is far from a general solution.

4. To some system administrators, honeypots are still hazardous components to be avoided. As a result, the use of honeypots may harm the deployability of host-based IPSs. The worries concerning the use of honeypots may be further deepened by the analysis performed after attacks are detected: in many host-based IPSs, buffered attack packets are replayed in the honeypot so that information about how vulnerable systems process those packets can be recorded.

In this dissertation, we propose the collaborative intrusion prevention (CIP) framework as a solution to the problems that spawned from the use of honeypots. Under the proposed framework, every production system will be equipped to detect and analyze attacks, and to generate and distribute countermeasures against them. In other word, every host under the collaborative intrusion prevention framework will play both the role of a protected system and that of the honeypot in traditional host-based IPSs.

With the capability of detecting and generating countermeasures for new attacks duplicated over a large number of production systems, the IPS will no longer have a single point of failure; the attacker can not evade the IPS by avoiding certain systems in the target network. Attempt to compromise any protected host may lead to the detection of the attack, and the production of countermeasure against the attack. Furthermore, each attempted attack will have the same chance of being detected. Thus, the more attack attempts are made, the higher the probability that a countermeasure will be made available to stop the attack. The threat of DoS against the honeypots also ceases to exist: a DoS that "blinds" our scheme will also make the target systems unavailable for being compromised. Finally, with every host being capable of detecting and generating countermeasures against attacks, the diversity in the software being run on the protected hosts is a much lesser issue. Even though rare applications are covered only by the few hosts running them, they are still not completely unprotected; on the contrary, in traditional IPSs, if there are no honeypots dedicated for these applications, attacks against them will

go unchecked.

The major challenge in implementing the idea of collaborative intrusion prevention lies in finding the right mechanism for detecting new attacks and collecting information about them. In fact, as we will see in Chapter 3, the lack of suitable detection/information collection mechanism is a general problem facing IPS design; existing mechanisms usually do not provide useful information and make countermeasure measure generation very difficult. To make the task of designing the CIP framework even more difficult is to make this detection/information collection component lightweight enough to run on production systems. In other word, to realize collaborative intrusion prevention, we need *a lightweight means to collect attack information which allows a very simple countermeasure generation process*. We solve the above problem by first identifying a class of intrusion detection systems (IDSs) that provide information to support simple countermeasure generation in the traditional, honeypot-based setting. After that, we modify our solution to use IDSs that allow the task of attack detection and information collection to be distributed in the following manner:

1. the overhead incurred by running the IDS can be made arbitrarily small
2. the information collected on any host that detects an attack will be sufficient for generating a countermeasure against the attack
3. while each collaborating host may have a very small chance of detecting a new attack, the probability that some host in the collaboration

will detect the attack before a non-trivial portion of the population are compromised is very high

4. each vulnerable host in the collaboration will have a similar chance of detecting a new attack, if they all run the IDS under the same configuration

We argue that the above properties will make our collaborative intrusion prevention simple, robust, with no single point of failure, and give incentive for hosts to contribute to the collaboration. We will elaborate on this point in Chapter 6.

The rest of this dissertation will be organized as follow: in the next chapter, we will highlight the importance of information collected from the attacked hosts by presenting our work on allergy attack. In the two chapters that follow, we'll present two foundations for building the CIP framework. In particular, in Chapter 3, we'll present our work on the LAIDS/LIDS framework, which solves the problem of collecting attack information in the traditional IPS setting; in particular, under the LAIDS/LIDS framework, once the underlying IDS detects an attack, countermeasures against that attack can be generated with no analysis at all (the process only involves retrieving a small amount of saved information). In Chapter 4, we'll present our work on random-inspection-based IDS, which is used for detecting and collecting information about new attacks in the CIP framework. In Chapter 5, we'll present our study of the false positive cases in our prototype random-inspection-based

IDS, and present a method to filter out all these false positives in real time. In 6, we'll give details of the CIP framework, as well as present how we use the proposed framework to build a prototype collaborative intrusion prevention system. Finally we'll conclude the work done in this dissertation in Chapter 7.

Chapter 2

Allergy Attacks: the Problem with Network-based IPS

In this chapter, we will present our work on allergy attack, which demonstrates the difficulties facing network-based IPSs and highlights the importance of information collected from the attacked hosts in the generation of effective and reliable countermeasures. We will start our discussion by defining "network-based IPSs".

In this dissertation, we define network-based IPSs as IPSs that observe network traffic to and/or from a protected network, identify attacks in the observed traffic and output signatures to filter out future instances of the attacks. For our discussion, the most important property of this class of IPSs is that they do not consider any information collected on the attacked hosts. In other word, the IPS does not know how the attack traffic is processed by its target, or even if what it considers to be attacks are really attacks. Such property brings the biggest strength and weakness of network-based IPS. The advantage of this purely network-based approach is that the same IPS can be used to protect a network of systems running different OSs and applications. Once installed, the IPS does not have to be reconfigured when machines are

added to the network. However, such easy management and high scalability comes at a price: network-based IPSs are found to be vulnerable to both attack polymorphism and allergy attack. Before we go into the details of allergy attack, we will present related work in the area of network-based IPS and work that investigate how attack polymorphism can be used to evade these IPSs.

2.1 Related Work

2.1.1 Network-based IPSs

Our literature survey shows that most existing network-based IPSs can be described by the following framework:

1. heuristic-based intrusion detection or honeypots are applied to identify suspicious traffic.

2. the results of the first step is analyzed to output the "best" signatures for filtering the suspicious traffic identified.

Since the main goal of many network-based IPSs is to output signatures for worm traffic, the heuristics applied in the first step invariably focus on worm-like behavior. For example, [42] classifies all traffic from an IP that has many failed connection attempts as suspicious, under the assumption that a worm will try to spread to many different addresses, many of which being invalid. On the other hand, [80, 49] models worm traffic as those that contain byte-sequences appearing in traffic that originates from, and destined to many

different addresses. An example of systems that use honeypots in the first step can be found in [45]. For these systems, any traffic destined to the honeypot is considered to be suspicious.

A point that's worth noting about the first step in the above framework is that many IPSs explicitly assume an imperfect detection that has non-zero false positives, and thus the IPSs are designed with some capabilities to guard against the case where signatures are generated for normal traffic that got misclassified in the first step. For example, both [80, 42] use a blacklist mechanism to prevent the IPS from generating signatures for traffic that the detector in the first step is known to misclassify. On the other hand, [64, 53] use a corpus of known normal traffic to guide the signature generation process, and avoid outputting any signature that causes a non-trivial amount of false positives.

As for the second step in the above framework, it is considered by many the main focus of research in network-based IPSs. The traditional approach was to output as signature one consecutive byte sequence that is prevalent in suspicious traffic. The idea behind this approach is to use byte sequences that correspond to the invariable parts of worm traffic as signatures, and thus gives the signatures generated some resistance to worm polymorphism. Examples of systems that employ this approach include [42, 80, 45, 91].

However, it is soon discovered that invariants in worm traffic seldom appear as a consecutive byte sequence that is long enough to be used as a signature without causing too much false positives. To solve this problem,

Polygraph in [64] proposed a token-based approach; in a preprocessing to signature generation, all distinct byte sequences that appear in at least K out of n samples in the collected suspicious traffic are extracted as tokens. After this token extraction, the signature generation process in Polygraph [64] will try to output sequences/sets of tokens as worm signatures. Polygraph can also return as worm signature a Bayesian classifier that computes the anomaly score of input traffic based on the occurrence of the different extracted tokens. Another IPS called Hamsa [53] employs the same token-extraction technique, but outputs worm signatures in the form of multisets of tokens.

Under the token-based approach, invariant properties of worm traffic can be described as the appearance of a number of short byte sequences. Due to the large number of possible signatures in the form of sequence/set/multiset of tokens, token-based systems usually employ some optimization techniques to find signatures that have high coverage in the pool of suspicious traffic (identified in the first step) and low occurrence in the normal traffic corpus.

Finally, another interesting approach is proposed in [49], where suspicious traffic identified in the first step are considered to contain executable code, and the signature generation process aims to identify control structure that is prevalent in such code carrying traffic as signatures. As such, signatures generated by [49] offer some resistance to worm polymorphism that does not change the control structure of the payload.

2.1.2 Attacking Network-based IPSs

Worm polymorphism has long been perceived as a potential problem facing network-based IPSs. In fact, this problem can be seen as the motivation for a lot of research in this area. For example, as mentioned before, the signature format in [49] provides the IPS with some defense against polymorphism. The token-based approach in [64] is also proposed to better capture the invariant part of worm traffic, and allows the IPS to ignore parts of the worm that attackers can modify. However, the work in [70, 65] show that even the most advanced network-based IPSs which employ the token-based approach can be evaded by carefully crafted attack traffic.

The basic idea behind the attack in both [70, 65] is to manipulate the token-extraction process, so that spurious tokens of the attackers' choosing will be used in signature generation. For systems that output signatures as sequence/set/multiset of tokens, the spurious tokens can cause the signature generation to ignore tokens that correspond to invariant parts of the attack, or output signatures that contain a lot of the spurious tokens. In the first case, the resulting signatures are entirely useless, and in the second case, the attacker can easily evade detection by not including the spurious tokens in future attack. As for the case where the IPS outputs Bayesian classifiers as worm signatures, the attacker will pick the spurious tokens in such a way that the resulting signature will have non-trivial false positives; in other word, it will be very hard for the signature to distinguish a real attack from benign traffic. The major difference between the attack in [70] and [65] is that in

[70], the spurious tokens are injected through "noise" traffic, i.e. non-attack misclassified in the first step of intrusion prevention, while in [65], such tokens can come as part of the real attacks.

As for the solution to the above problems with attack polymorphism, the authors of [70, 65] both suggest a more semantic-based approach for intrusion prevention. In particular, the authors of [70] advocate the use of semantic-aware sensors to identify attacks, and thus avoid including noise traffic in the signature generation process. On the other hand, the authors of [65] argue that information about how the target software process malicious inputs will be a defense against the attack they describe. We see that both these conclusions (especially the latter) support our view; a purely network-based approach will be insufficient, information collected on the attacked hosts are essential to an effective and reliable IPS.

2.2 Allergy Attack

Though attacks in [70, 65] shows how even the most advanced network-based IPSs can be rendered useless by carefully designed attacks, our work in [12, 13] demonstrate that the problem can be much more severe. In particular, we've identified a kind of attack that we called the allergy attack, which allow attackers to manipulate the vulnerable IPSs to filter out normal traffic of their choosing, and turn these IPSs into active threats to the protected network. We defined the allergy attack as follow:

> An allergy attack is a denial of service (DoS) attack achieved through inducing network-based IPSs into generating signatures that match normal traffic. Thus, when the signatures generated are applied to the perimeter defense, the target normal traffic will be blocked and result in the desired DoS.

The problem with allergy attack is similar to the "causative, indiscriminate availability" attack mentioned in [4]. However, the work in [4] focuses on the much higher level problem of attacking machine-learning based security mechanisms in a theoretical setting. While the authors of [4] have proposed many different means of abusing a machine-learning based system (e.g. inducing high false positives, evading detection), our study of allergy attacks is more specific on one particular issue, the viability of inducing high false positives in a network-based IPS in practice. Also note that our work in [12] is the first to extensively study this particular issue in the context of network-based IPSs; before that, the possibility of an allergy-type attack has only been briefly mentioned in [42, 80, 98]. The most detailed documentation of this potential problem can be found in [80]. We quote from Singh et al in [80]:

> Moreover, automated containment also provokes the issue of attackers purposely trying to trigger a worm defense - thereby causing denial-of-service on legitimate traffic also carrying the string.

In fact, the work in [80] is the only one that has explicitly mentioned the possibility of denial-of-service resulting from an allergy-type attack. In

[42], the problem is referred to as "attackers deliberately submit innocuous traffic to the system", while Yegneswaran et al used the term "intentional data pollution" in [98].

2.2.1 Implementing Allergy Attacks: Overview

In the following discussion, we'll focus on the allergy attack against valid requests for services provided by the attacked network. Upon a successful attack, the signatures generated will block all instances of the target requests at the perimeter defense, and make the corresponding service unavailable to the outside world. As will be seen, allergy attacks allow the attacker to have very fine grain control over what services to be attacked (e.g., instead of blocking access to the entire web-site, the attacker may choose to make only particular pages unavailable). Furthermore, we note that it is also possible to have allergy attacks against responses generated by servers providing the targeted service, and the design will be very similar to what we are going to present. However, the result of such attack will be less devastating to the service provider, and more confined to the clients within the attacked network, and we'll leave the study of this flavor of allergy attack to future work.

Based on the two-step framework for network-based IPSs presented in Setc. 2.1, the implementation of allergy attack should focus on crafting attack packets with the following properties:

1. The packets will be classified by the vulnerable system as suspicious, and will be used for signature generation.

2. The packets, when used for signature generation, will result in the desired signatures being generated.

Note that even though many network-based IPSs have means to handle harmful signatures generated for traffic mistakenly identified as suspicious in the first step of the framework in Sect. 2.1, most of these systems remain vulnerable to allergy attack. This is mainly because the mechanisms employed are designed to tackle "naturally occurring" false positives, not those intentionally produced by the attackers. Furthermore, as we'll show in Sect. 2.3, allergy attack is still possible even if we check each new signature against some corpus of normal traffic.

2.2.2 Allergy Attacks: Against Autograph

In this section, we shall demonstrate the allergy attack against an example network-based IPSs, namely Autograph [42]. We note that Autograph is a typical example of the first generation of network-based IPSs, and the allergy attacks against other network-based IPSs from the same time period are largely the same as that presented below. To give some background about how Autograph works, a brief description of Autograph is given in Appendix A. For the sake of completeness, we will also present in Appendix B the technique we developed in [12] for defeating the blacklisting mechanism used in Autograph (which guards against signatures mistakenly generated for benign traffic). We note that the technique in Appendix B is specific to Autograph. In the next section, we will present general techniques to defeat any attempts

to stop allergy attacks by vetting signatures against known good traffic.

Our attack against Autograph is divided into two steps. In the first step, we induce Autograph into classifying the machines we control (our "drones") as scanners, so that any traffic from our drones will be considered suspicious and will be used in signature generation by Autograph. In the second step, we simply use the drones to connect to machines in the protected network, and populate Autograph's suspicious pool for the target port with our attack packets. These packets are crafted such that the desired signatures will be generated when they are used for signature generation. To ease our discussion in this and the next section, we assume the target traffic to be an HTTP request for a protected web server. We stress once again that other types of traffic can also be targeted, and HTTP requests are chosen only because it appears to be the most direct way to inflict loss to the attacked organization.

Due to the simple scanner detection heuristics used by Autograph, the first step can be easily achieved by requesting connections with many random IP addresses. For some networks, an easier and faster method is available; we can send out TCP connection requests with a combination of flags that never appears in normal traffic (e.g. with both SYN and URG set). Since these requests are dropped or rejected by most networks, they will be considered failure by Autograph. Thus our drones will be classified as scanners with very few packets sent. This latter technique is actually employed in our experiments.

After being classified as a scanner, each drone will proceed with the

second stage where crafted attack packets are sent to the target network over successful TCP connections. If no parts of the target request are blacklisted, we can simply put the entire target request in our attack packets. *Since the experiments in [42] show that Autograph has very low false positives, we believe it is very unlikely for any part of the target request to be blacklisted. In other word, our simple allergy attack will succeed most of the time.* Furthermore, our experiments (see Appendix B) show that even if content blocks from the target requests are all blacklisted in the training phase, we can still achieve a successful allergy attack after a small number of trials, and this result holds for all the HTTP requests we've tested, over a reasonable range of configurations for Autograph.

At this point of our discussion, it is tempting to think we can stop allergy attacks if we keep a corpus of known benign traffic, and only deploy those signatures that do not match too much traffic in the corpus. In fact, token-based IPSs like Polygraph and Hamsa [64, 53] assumes the availability of such a corpus, and use the corpus to guide the search for signatures that have both high coverage for suspicious traffic, and a low expected false positive rate. However, we find that the evolving and diverse nature of normal traffic renders such "corpus-based" defense ineffective against allergy attack. In the following, we will elaborate on this point and present two advanced forms of allergy attacks that can evade a corpus-based defense. We call these advanced allergy attacks "type II" and "type III" attack.

2.3 Type II Allergy Attack

The term "type II allergy attack" was coined in [12] as a specific type of allergy attack, though the idea first appeared in [80] as a threat against their blacklisting mechanism, quoted as follows:

> However, even this approach may fall short against a sophisticated attacker with prior knowledge of an unreleased document. In this scenario an attacker might coerce Earlybird into blocking the documents released by simulating a worm containing substrings unique only to the unreleased document.

In other word, the type II allergy attack targets future traffic and induces the network-based IPS into generating signatures to match patterns that appear in future traffic, but not those at present. As a result, the generated signatures will be deemed acceptable when matched against the blacklist in [80, 42], or any static corpus which cannot predict what future traffic will be like. In order to prevent type II attacks, the defender must identify all traffic components that evolve over time (and avoid generating signatures for those components), or the signatures must be constantly re-evaluated[1].

A point worth noting is that it is not always necessary to predict how traffic will evolve in order to launch a type II attack. The discussions in [80, 12] assume that the corpus is always "fresh" and captures all the normal traffic

[1] there are simply too many events that can change normal traffic to practically enumerate them and perform the checking only when these events occurs.

at the time of the attack. However, it may not always be feasible to keep an up-to-date corpus; in addition to the possibly prohibitive cost of constantly updating the corpus, as mentioned in [64], a relatively old corpus may also be needed as a defense against attempts to contaminate the normal traffic corpus (the authors of [64] called it "innocuous pool poisoning"). In particular, [64] pointed out that if attackers can introduce into the corpus traffic that contain tokens which make up the invariant parts of an imminent attack, they can thwart the signature generation by making all candidate signatures appear to have high false positive rate. The authors of [64] suggest that the use of an old normal traffic corpus will introduce a time delay between the poisoning of the corpus and the real attack, and hopefully this time will allow any vulnerabilities that attackers are targeting to be patched.

In other word, instead of targeting "future" traffic only, we should consider a type II allergy attack as one that induces the target IPS into generating signatures to filter traffic that appears only after the corpus is generated. As we will see, this significantly increases the power of the type II allergy attacks, and allows the attack to have instant effect.

In the following, we will show how some components common in HTTP requests can be exploited by a type II attack, and analyze the amount of damages that these attacks can cause on some example web sites.

2.3.1 Dates in URLs

The first common component in HTTP requests that can be utilized by a type II allergy attack is the date encoded in URLs. Websites that constantly put up new materials while keeping old ones available usually have the creation date of a page encoded somewhere in its URL. This provides a very handy way of organizing materials created at different time. Examples of websites that organize their pages in this manner include CNN.com, whitehouse.gov, yahoo.com and symantec.com.

In the following, we will use CNN.com as an example and see how type II attacks targeting dates encoded in URLs can affect visitors to CNN.com. We quantify the damage done by our attack with a metric called the "broken link probability" (BLP) which measures the probability that a visitor to the attacked site (CNN.com) will try to access a page made unavailable by our attack. For the computation of BLP, we use a localized version of the random surfer model from [68]. Under the localized random surfer model, a surfer always starts his/her visit to the attacked site at the "root page" (in this case, www.cnn.com). At each page visited, the surfer will randomly follow links on that page with a probability d=0.85, or "get bored" and leave the site with probability 1-d (i.e. 0.15). The BLP can then be computed as the sum of the page rank (computed under the localize random surfer model) for pages made unavailable by our attack, since page rank of a page can be seen as the probability that a visitor will follow a sequence of links that lead to the page under concern (the formal definition of BLP, as well as the actual algorithm

for computing it can be found in Appendix C). Also note that since all traffic generated during a user's visit will belong to the same TCP flow, we can use the BLP as an estimate of the false positive rate resulting from an allergic signature when it is evaluated against a normal traffic corpus. In particular, any TCP flow in the corpus that got filtered by some allergic signature will correspond to the visit of some user, one in which the user visits the same sequence of pages as in the flow until the first unreachable page is accessed. Thus, if the traffic in the corpus is generated by visitors that follow (more or less) our localized random surfer model and if the corpus is large enough, the BLP will be a good prediction of the portion of flows that matches an allergic signature.

We start our study of CNN.com by finding out URLs of pages under CNN.com, as well as how they link to one another. For this purpose, we employ a simple web crawler based on [58]. Our web crawler starts at www.cnn.com, the "root page" under the localized random surfer model. Because of resource limitation, we only focus on pages that are reachable within 5 clicks from the root page. Furthermore, at any visited page, the crawler will only expand its exploration to pages that either reside in the same directory as the current page, or are in a direct subdirectory of the one holding the current page. However, due to the redirection of some URLs under CNN.com to other sites, our web crawler also collects information of pages under Time.com, EW.com and Money.cnn.com. We performed our experiments from 16th Feb to 9th Mar, 2007, and crawled the target site at 9am and 12 noon every day. In

all our experiments, the web crawler retrieved more than 5000 URLs in total, and more than 1000 of the URLs are under the server CNN.com. We note the above restrictions may result in undercounted BLP for some allergic signatures. However, since pages that are more than 5 clicks away from the root usually have very low page rank, and pages under CNN.com usually link to other pages that are either in the same directory or a subdirectory, we believe the inaccuracy caused by the restrictions on the web crawler should be minimal.

With the information collected, we studied how the BLP of 5 signatures that encode the date of 24th to 28th Feb evolve from 5 days before to 4 days after the designated day (e.g. for the signature "/02/24/", we measured its BLP for each of the two data sets collected from 19th to the 28th of Feb). As mentioned before, we use the BLP as both a measure of the damage caused by the allergic signature and an estimate of the false positive caused when it is evaluated against traffic collected on a particular day. Finally, in the following discussion, we will call the day designated by the "date-encoding" signature "day 0", the day that's one day before will be denoted as "day -1", that which is one day after "day 1", and so on. The results of our experiments are shown in Fig. 2.1 and 2.2.

As we see from Fig. 2.1, all 5 tested signatures produce a zero BLP before the corresponding day 0. We have experimented with other allergic signatures which encode the dates ranging from 16th Feb to 9th Mar, and they all show a similar pattern. Though in some cases, the byte sequences targeted by the tested allergic signatures appear before the corresponding day

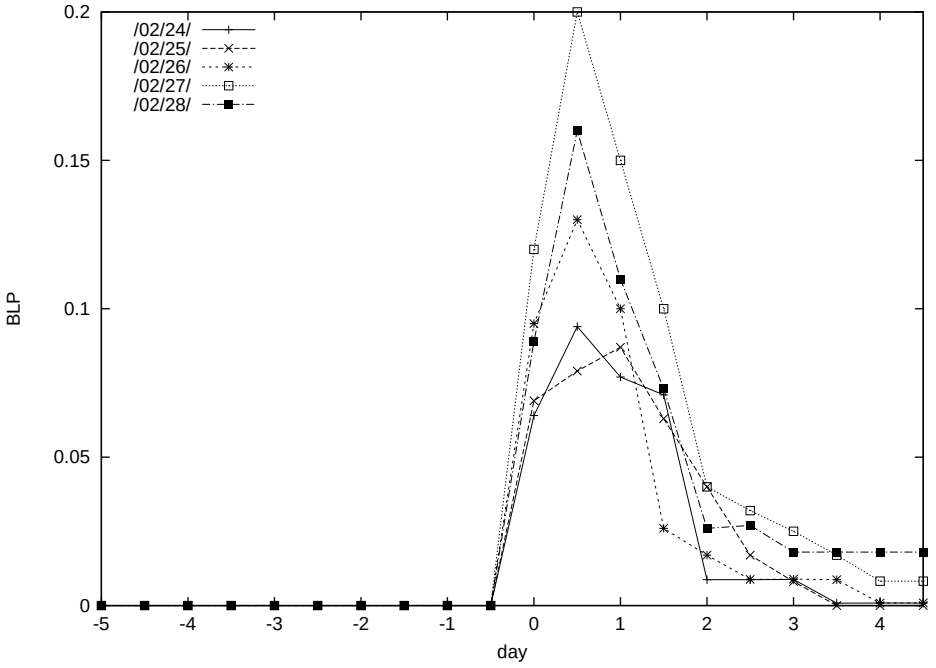

Figure 2.1: The BLP of 5 different date-encoded signatures changes from 5 days before to 4 days after the designated date (with the designated date denoted by day 0, days before that denoted by day -1, day -2 and so forth, days after are denoted day 1, day 2, etc). The BLP of the tested signature at 9am of day n is denoted by the point directly above the mark "n" on the x-axis, while the BLP at 12noon is denoted by the point between "n" and "n+1" on the x-axis.

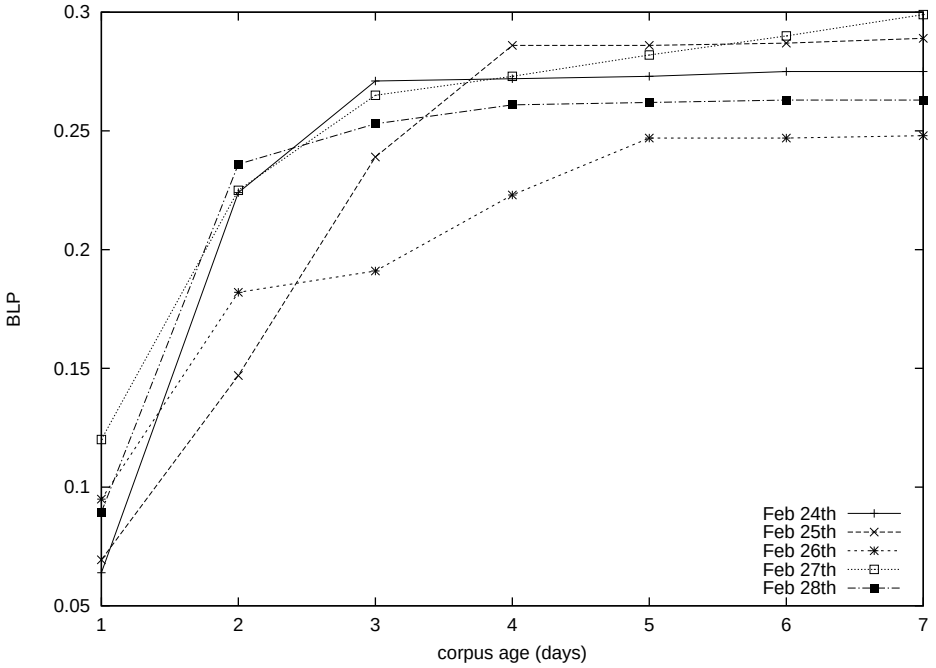

Figure 2.2: The effectiveness of type II attacks that target dates in URL, measured in BLP, when used against corpus of different age and launched on 5 different days (24th - 28th Feb).

0. This is usually caused by URLs that point to pages created in the previous years (e.g. we find the string "/02/21/" in two URLs that point to the 21st Feb, 2005 issue of the Money magazine). Nonetheless, the BLP of all the tested signatures remain below $1.5 * 10^{-6}$ before day 0. Thus, any allergic signature encoding a date after the corpus is generated will have a false positive below $1.5 * 10^{-4}\%$ when evaluated against the corpus[2]. In other word, the type II allergy attack that employ "date-encoding" signatures will evade even corpus-based defenses with a very low false positive threshold (both [70, 53] suggested a 1% threshold, while the lowest threshold used in [64] is 0.001%).

Now let's consider the power of the described attack against an up-to-date corpus. Assuming that any allergic signature will be removed within a day since it start filtering normal traffic, it appears the attacker should induce the IPS into generating one single allergic signature for some future day (extra signatures will take effect on a different day, and thus cannot add to the damages at day 0). From Fig. 2.2, we see that this attack will create a more than 6% chance for visitors to CNN.com to reach an unavailable page if the allergic signature is not removed by 9am. Also, note that the two days with the lowest BLP, 24th and 25th Feb, are both weekend days. In other word, the amount of damage for the type II allergy attack studied above can be far greater if it targets a weekday; the BLP created can be as high as 0.12 at 9am, and up to 0.2 if the attack is not stopped by noon. Finally,

[2]We believe it is highly unlikely that the studied signatures will match some other parts of an HTTP requests, since dates in other fields are represented differently, and the use of "/" outside the URL is very uncommon.

we'd like to point out that the attack against an up-to-date corpus requires a certain "build-up" time to reach the level of damage predicted; i.e. the figures given above only apply if the attack is not detected until 9am or 12noon. If the allergic signature is removed in the first few hours of day 0, the damage caused will be much smaller.

On the other hand, if the corpus is n-day old, with the same notation used above, the attacker can induce the IPS to generate signatures for the date of day 0 to day -(n-1). For example, the attack on 16th Feb against a 3-day-old corpus will involve the signatures "/02/16", "/02/15/" and "/02/14/". We have experimented with the effectiveness of this attack when it is launched at noon of the 5 different days tested above, against a corpus of "age" ranging from 1 day to a week, the results of our experiments are shown in Fig. 2.2.

As shown in Fig. 2.2, the use of a 2-day-old corpus instead of a fresh one will almost double the damage caused by the attack, and an attack against a one-week old corpus will produce a BLP of 0.25 to 0.3 with just 7 signatures. Thus, the attack against an old corpus is significantly more powerful than that against a "fresh" one. Furthermore, by targeting existing traffic patterns, the attack can produce instant effect; in other word, the BLP resulted will reach its maximum once the allergic signatures are in place. This is a sharp contrast to the attack against a "fresh" corpus which may take a few hours to build up its level of damage.

Finally, we note that the attacks described above are easily identifiable once the broken links are reported and human intervention is called in.

Nonetheless, such need of human intervention defeats the purpose of IPSs, and the attacks can make some important parts of the target site temporarily unavailable.

2.3.2 Timestamp in Cookies

Another component in HTTP traffic that can be utilized by a type II attack is the timestamp in web cookies. Web cookies are employed by many sites to keep track of user preferences. New visitors to these websites will receive a set of web cookies together with the content of the first page requested. The cookies will be stored in the user's machine, and will be sent with all further HTTP requests to the site. Also, an expiration date is associated with each cookie sent to the user, and when the expire date is reached, a new cookie will be issued.

We find that some sites use cookies to record the time for various user events. For example, cookies from Amazon.com contains an 11-digit "session-id-time" which expires in a week and records the day where the user's last session started.

A type II allergy attack can exploit this timestamp cookie by inducing the IPS into generating signatures that match future values taken by these cookies (or their prefixes). To avoid the signatures from unintendedly matching other parts of HTTP requests, the name of the cookies should be included, i.e. the signature should be of the form "session-id-time=xxxx". With this signature format and a value for "xxxx" that is only used after the corpus is

generated, the signatures should be deemed usable by the IPS.

As for the effectiveness of the attack, let's assume the corpus used is up-to-date. The attack will then employ a signature that filters the value taken by the "session-id-time" cookie on a particular future day 0, and will make all pages under Amazon.com inaccessible to any user who has the corresponding cookie expires on or before day 0; their session-id-time cookie will be updated to the value targeted by the attack after the first request, resulting in all subsequent requests being filtered. On the other hand, if the IPS employs an old corpus, the attack can target all values that the timestamp cookies can take after the corpus is generated, and create more significant damages. Note that virtually all HTTP requests to Amazon.com will contain a "session-id-time" cookie that is generated between day 0 and day -6; any other timestamp cookies will have expired, and will be updated after the first request. As a result, if the corpus used is more than one week old, the attacker can induce the IPS into generating signatures for all valid values of the "session-id-time" cookie, and effectively make all pages under Amazon.com unavailable.

In conclusion, an up-to-date corpus is very effective in limiting the power of a type II attack. However, using a "fresh" corpus also makes it easier for worms to evade the IPS through innocuous pool poisoning. The use of a corpus with traffic collected over a long period of time (which is a solution to "innocuous pool poisoning" proposed in [53]) may have the same effect as using an old corpus. Let's consider the encoded-date attack in Setc. 2.3.1 against a corpus with traffic collected over a month (i.e. from day 0 to day

-30). At 12noon of day 0, we can assume that the allergic signature encoding the date for day 0 to appear in 20% of the traffic for that day, but appears in close to 0% in the remaining 30 days of traffic in the corpus. Similarly, the byte sequence that encodes the date for day -1 will appear in 20% and 10% of traffic on day -1 and day 0 respectively, and never appear for the other days. As a result, both signatures will match less than 1% of all the traffic in the corpus, and can be used in a type II attack to create a BLP of 0.15 to 0.2. Further analysis shows that the sum of the BLP at noon from day 0 to day 4 is at most 0.36 for the 5 signatures tested in Sect. 2.3.1. Thus, a corpus with over 40 days' traffic will probably allow allergic signatures for the date of day 0 to day -7 to be used to create the same level of damage as when the type II attack is launched against a one-week-old corpus.

2.4 Type III Allergy Attack

A more nuanced weakness of a corpus-based defense is the diversity in normal traffic, which is exploited in a type III allergy attack. We define a type III attack as follows:

> A type III allergy attack is an attack that induces the target IPS into generating a set of signatures, such that each will have a false positive low enough to be acceptable to the IPS, but as a whole, the set will block a significant portion of normal traffic and amount to a non-trivial DoS against the target network.

The main difference between the type II and the type III attack is that signatures generated by the former have their false positives increase significantly over time, while false positive rates for signatures from the latter stay at a low level. In other word, the type III attack takes a more "brute-force" approach, and requires more signatures than the type II attack. On the other hand, the type III attack is much more flexible, and is much easier to design.

We can also see the type III attack as a divide-and-conquer strategy; it "divides" the target traffic into small pieces, and "conquer" each with an allergic signature specific for that piece. With signatures specific for small pieces of traffic, we can guarantee that each signature will have a sufficiently low false positive. However, the success of this strategy depends on the following conditions:

1. The IPS must tolerate signatures that cause some minimal false positives.

2. There must be sufficient diversity in the normal traffic for the attacker to "divide" them into small pieces, each distinguished by the signature that matches only that piece but nothing else. In other word, if there is very little variation among normal traffic, any allergic signature will have a very high false positive, and it would be impossible to launch a type III attack.

Our literature survey shows that the first condition should be met by any reasonable network-based IPS; in order for the IPS to be of any use, it

must tolerate a certain degree of false positives in the signatures. This is because the corpus may contain anomalous traffic, even after all instances of known attacks have been removed. In fact, the studies in [70] found that 0.007% of traffic in their corpus matches the signature for the true invariant bytes of the worm they've tested. The author of [70] also reported a similar 0.008% of anomalous traffic in the innocuous pool used in [64]. In other word, if the network-based IPS were to be effective against the worm tested in [70], it must accept signatures that match as much as 0.008% of flows in the normal traffic corpus. For our discussions below, we assume the IPS will accept any signature that matches less than 1% of the traffic in the corpus[3]. Next, let us consider how the attacker can "divide" the normal traffic and satisfy the second condition.

2.4.1 Diversity in Pages Visited

Once again, let's take CNN.com as our example target. For any website of the size of CNN.com, the BLP of a page may drop very quickly with the number of clicks required to reach that page from the root. In other word, pages which are only reachable after 2 or 3 clicks from the root page may well have BLP far below 0.01, our false positive threshold. This is especially true for CNN.com, where pages tend to have a large number of links (e.g. the root page alone points to more than 100 pages). Thus, the mere size of our target site guarantees the diversity needed for a type III allergy attack; all but the

[3]Both [53, 70] use a false positive threshold of 1%.

most popular pages under these sites are requested only in a very small portion of user sessions. As a result, an allergic signature that targets requests for any particular page is very likely to evade a corpus-based defense, and a significant amount of damage can be caused by a large number of such signatures, each matching requests for different pages. In the following, we will present the results of attacking CNN.com with the type III allergy attack.

We construct our attack against CNN.com with a very generic method that can be applied to any other website. In particular, we search over all pages under our target site, starting with the root page, and consider pages reachable with fewer clicks from the root first. For any page examined, we compute the BLP expected if that page is blocked. If the BLP is lower than the threshold, we mark that page as a target, otherwise, we "expand" the search from that page (i.e. examining all pages pointed to by the current page later). For each target page, we extract random 10-byte subsequences from the "path" part of its URL, and use the first one with BLP below the threshold as the allergic signature for that page. Finally, we sort the signatures in descending order of their BLP, and compute the total BLP resulted when different number of these signatures is applied. We have repeated this experiment for the five data sets collected at 9am of 24th to 28th Feb, and the results are shown in Fig. 2.3.

As we can see, the first 50 allergic signatures always create a BLP of more than 0.25, and an additional 50 signatures will bring the BLP up to 0.6. Also note that the algorithm presented is not optimized for finding the

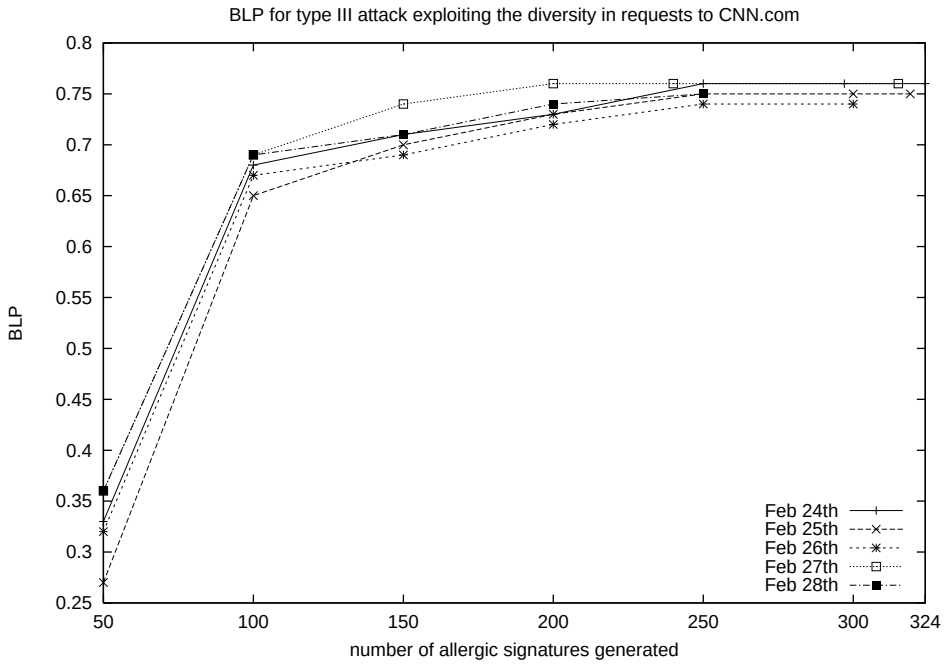

Figure 2.3: BLP caused by different number of allergic signatures from the type III attack targeting the "not-so-popular" pages under CNN.com.

smallest set of signatures that creates the maximum BLP; instead, it is only intended as a simple proof-of-concept. Thus, it is entirely possible for a type III attack to produce the same level of damage predicted in Fig. 2.3 with fewer signatures.

2.4.2 Diversity in Search Terms

The diversity of keywords queried at different search engines like Google.com can also be exploited in a type III attack. We conjecture that the queries from different users are so diverse that even the most frequently searched keywords are involved in a very small portion of flows, and the data from Hitwise [33] seems to support this conjecture. By collecting network data from various ISPs, Hitwise provides various statistics concerning the use of search terms at various search engines. According to Hitwise, the top 10 search terms "that successfully drove traffic to websites in the Hitwise All Categories category for the 4 weeks ending February 24, 2007, based on US Internet usage" are as shown in Table 2.1.

As we can see, even the most popular keyword, "myspace" accounted for only 1.07% of all observed searches. Furthermore, the volume of searches received drops quickly with a search term's ranking. Even though it is not clear how Hitwise come up with their ranking, the data above seems to suggest that all but the most popular search terms will appear in a far less than 1% of traffic. Thus, an allergic signature targeting queries for a specific search term will most likely have a false positive low enough to evade any corpus-based

Rank	Search Term	Volume
1	myspace	1.07%
2	myspace.com	0.64%
3	ebay	0.41%
4	www.myspace.com	0.35%
5	yahoo	0.21%
6	mapquest	0.18%
7	myspace layouts	0.18%
8	youtube	0.18%
9	craigslist	0.14%
10	yahoo.com	0.14%

Table 2.1: Top 10 search terms for the 4 weeks ending 24th Feb, 2007, with the percentage of searches that each term accounts for.

defense.

Even though it is hard to evaluate the power of an allergic signature that blocks out all queries for a particular search term, we argue that the damage caused by such attacks can be non-trivial and many-folded. First of all, this may mean direct business loss to the search engine. Let's take Google.com as an example. Under Google's advertising program, Google AdWords, each advertisement is associated with a set of search terms, and it only appears when a user searches for one of those terms. Furthermore, Google only charges an advertiser when a user clicks on his/her advertisement. As a result, a type III attack that blocks out all queries for search terms associated with an advertisement will make that advertisement completely non-profitable for Google.

The type III attack described above will also affect parties whose web-

sites will be listed when somebody queries on the targeted keywords. The most obvious example victims are the advertisers on Google AdWord whose advertisements will never reach their customers. Damages can also come in other flavors. For example, according to [85], the following search terms: "BARACK OBAMA", "HILLARY CLINTON" and "JOHN EWARDS" (three politicians running for the president of the US in the 2008 election) all accounts for less than 0.01% of all searches observed by Hitwise between Sep 2006 and Jan 2007. In other word, it is entirely feasible to have a type III attack that blocks out all searches for a particular candidate, which may create non-trivial damage to his/her campaign.

2.4.3 Cookies Revisited

In addition to recording time, web cookies are sometimes used to distinguish different users/user sessions. For example, the cookies from Google.com include a 16-digit hexadecimal value called "PREF-ID", which uniquely identifies a user. Similarly, both Yahoo.com and Amazon.com include an ID for either the user or the corresponding user session in their cookies. The uniqueness of these "ID cookies" are introducing the diversity necessary for type III attacks into normal traffic, and can be exploited as follow: suppose the target cookie can take n values in each byte/digit, we will generate one allergic signature to match each of the possible values taken by the first k bytes/digits of the cookie, with k being the smallest integer such that $1/n^k$ is below the false positive threshold. To make sure that each signature only matches the

beginning of the cookie value as intended, we will include the name of the target cookie as well.

For all the "ID cookies" we have seen, their values remain the same throughout a user session. Thus, each flow in the corpus will match exactly one of the allergic signatures. Furthermore, the values of these "ID cookies" are usually assigned such that the portion of cookies starting with a certain byte sequence is the same as the portion with any other prefix. As a result, each of the above allergic signatures will have a false positive very close to $1/n^k$, and thus will evade the corpus-based defense. Finally, since the allergic signatures cover all possible prefix of the target cookie, they will filter out almost all traffic to the target site.

We have experimented with the above attack by collecting 10 sets of cookies from Google.com, with 100,000 cookies in every set. We measured the distribution of the values for the first two bytes of the "PREF-ID" cookie, and find that each two-byte prefix of "PREF-ID" appears in 0.47% to 0.33% of cookies in each data set. In other word, the described type III attack allows us to evade a corpus-based defense with a threshold of far less than 1%, and virtually block all traffic to Google.com with 256 signatures.

We note that the type III attacks will be much less effective if a lower false positive threshold is used. For example, if the threshold is lowered to 0.01% (which appears to be the lowest possible value according to [70]), we find that the attack against CNN.com described in Sect. 5.1 will require more than 1000 signatures to achieve a BLP of less than 0.02. The attack based on the

diversity in search terms may be less affected by a lower false positive threshold, since the figures from Hitwise seem to suggest that there are plenty of search terms that appear in less than 0.01% of traffic, and a significantly larger set of signatures may be required for the attack targeting the "PREF-ID" cookie to block out all traffic to Google.com. However, a lower false positive threshold will also reduce the cost of evading the network-based IPS through innocuous pool poisoning: the attackers now need a much smaller volume of bogus traffic to make a real signature against their attack dropped by the corpus-based mechanism. In other word, the tradeoff between defending against allergy attacks and innocuous pool poisoning manifests itself once again. Finally, we will argue that the (possibly) large number of signatures involved in a type III attack is not necessarily a shortcoming. It gives the attack certain stealthiness: it would be hard to manually remove all the allergic signatures involved. A slow type III attack may also mean a constant influx of allergic signatures, each causing minor damages, which makes stopping the attack serious nuisances.

2.5 Experimenting With Polygraph and Hamsa

In this section, we will present our experience in launching the attacks described in Sect. 2.3.1, 2.4.1, and 2.4.3 (which target encoded dates and requests for less popular pages under CNN.com, and the identification cookie used by Google.com respectively) against Polygraph [64] and Hamsa [53] (a brief introduction of how these two IPSs work can be found in Sect. 2.1, and we'll refer readers interested in the details about these systems to [64, 53]). We

choose to experiment with these two systems because they are two of the most advanced network-based IPSs that limit their false positives with a corpus-based mechanism. We based our experiments on a slightly modified version of Polygraph provided by the authors of [70], and our own implementation of Hamsa. Our implementation of Hamsa deviates from that presented in [53] slightly: we do not require a token to appear in 15% or more of the worm flows in order to be used in the signature generation. We believe this requirement allows the attackers to evade the IPS easily, given that the attacker can always introduce noise as in [70], and some of the "invariant" parts of a worm may actually vary (e.g. in a stack buffer overflow, the return address can be over written with many different values). We note that the tested attacks should also be effective against the original Hamsa; we only need to carry them out in multiple rounds, each generating 6 allergic signatures.

The exact design of our attacks is very similar to that presented in Sect. 2.2.2; the only difference is what byte sequences we use to populate the suspicious pool of the attacked IPS. Thus in the following, we'll only focus on the effect of using various byte sequences in this second part of our attack.

2.5.1 Type II and Type III Attack against CNN.com

We have experimented with launching the two attacks against CNN.com on the same 5 days as studied in Sect. 2.3.1 and 2.4.1 (24th - 28th Feb, 2007). For the experiments on the type II attack, we generate a 7-day-old corpus

by simulating 50,000 user sessions[4] with the data collected 7 days before the corresponding day 0 (e.g. the experiment on the attack on 24th Feb uses a corpus generated from data collected on 17th Feb). For the type III attack, we assume a "fresh" corpus with 50,000 simulated user sessions based on the data collected at 9am of day 0. As for the worm pool, in our experiments on Hamsa and the conjunction/token-subsequence signature generator of Polygraph, we construct it to contain 3 copies of each allergic signature we want the IPS to generate. After that, we invoke the tested signature generation process once. We then evaluate the false positive caused by the generated signatures with 150,000 simulated user sessions generated using the data collected at 9am of the tested day 0. We find that the measured false positive rate from the type II attack is always within 1% of the computed BLP value. As for the type III attack, the false positives measured in the experiments are lower than predicted, but the difference is always within 6.2%.

The setup for the experiments on the Bayes signature generator in Polygraph is a little different, since the Bayes signature generation algorithm effectively generates one signature to cover all traffic in the worm pool, and guarantees that this "combined" signature has a false positive rate below the threshold. As a result, we may need to invoke the signature generation process multiple times to achieve the level of damages expected. Our experiments show that one invocation is sufficient for the tested type II attack, since the byte sequences involved in the attack rarely appear in the corpus. On the other

[4] [64] used a training set and testing set of 45,111 and 125,301 flows respectively.

hand, the type III attack requires multiple invocations of the Bayes signature generation process. Thus, we modify our experiment as follows: in each round of the experiment, we construct the worm pool with 5 of the target byte sequences that are not yet covered, 3 copies for each. We find that a little less than 100 rounds is needed to have all the target byte sequences filtered. As before, we evaluated the signatures generated for the two attacks with 150,000 simulated user sessions, and find the false positives obtained from the experiments are within 2% range of that predicted by our BLP analysis.

The discrepancy between the measured false positive and that predicted by the BLP analysis may be explained by the randomness in the generation of the corpus and the test traffic pool. The former may result in some target signatures matching more flows in the corpus than allowed, and prevent their inclusion in the final set of signatures. We believe this is the main reason why the measured false positives of the attacks against Hamsa and the conjunction/token-subsequence signature generation in Polygraph is 5% lower than expected. On the other hand, the fluctuation in the generation of the testing traffic pool affects the measured false positive rate of the generated signatures, which may account for the smaller differences seen in the other experiments.

2.5.2 Type III Attack against Goolge.com

For the type III attack targeting identification cookies from Google.com, we repeat the experiment 5 times. In each experiment, we construct the corpus

used by the IPSs with a different set of 50,000 cookies. The rest of the experimental set up is the same as above; i.e. we invoke the signature generator once with the worm pool containing all the target byte sequence for the experiments with Hamsa and the conjunction/token-subsequence generator of Polygraph, and perform the experiment in multiple rounds, each with 5 remaining target byte sequences for the Bayes signature generation. The generated signatures are then evaluated with 5 different sets of 100,000 cookies. The signatures generated result in a 100% false positive against the tested sets of cookies as expected. Once again, the attack against Hamsa and the conjunction/token-subsequence generator of Polygraph needs only one invocation of the signature generation process. On the other hand, the attack against the Bayes signature generation requires around 130 rounds to finish.

Obviously, the possible need to invoke the signature generator multiple times is a drawback of the type III attacks in general. Depending on the frequency at which the signature generation process can be invoked, the attack can take a long time to complete. Nonetheless, in order to contain fast propagating worms, the maximum time between two invocations cannot be too long; in [42], this is given as "on the order of ten minutes". Now, let's assume the signature generation can be invoked every 10 mins[5]; it will then

[5]According to [61], if content filtering is deployed under the "top 100 ISPs" scenario, a reaction time of 10 mins is necessary to protect 90% of vulnerable hosts against a worm capable of making 40 probes/sec, and the probe rate of Code-Red v2 is assumed to be 10/sec. Also note is that an invocation of the signature generation process every 10 mins is certainly insufficient in stopping SQL Slammer, which infected 90% of vulnerable hosts in 10 mins.

take around 8 hours to generate the top 50 allergic signatures in the type III attack against CNN.com (which will result in a BLP of more than 0.25).

2.6 Conclusions for Allergy Attack

In this chapter, we presented our study of allergy attacks against network-based IPSs. We found that many of these systems (in particular [42, 49, 80, 45, 53, 64]) can be "tricked" by a clever attacker into generating signatures that match a significant portion of the normal traffic destined to the network these IPSs are intended to protect. As a result, IPSs vulnerable to allergy attacks can be turned into active agents of DoS against the protected network. We have also shown that allergy attacks cannot be stopped by checking each generated signature against a corpus of known benign traffic. This is because no static corpus can predict what normal traffic in the future will look like, and thus cannot identify "allergic signatures" that target patterns that are yet to appear in normal traffic. Furthermore, such a corpus-based defense also cannot distinguish signatures matching very specific but benign traffic from those identifying true invariant properties of real attacks that happen to have non-zero false positive rate. As such, we found that even some of the most advanced network-based IPSs (namely Polygraph and Hamsa) are vulnerable to what we called type II and type III allergy attacks. Our experiments show that when Polygraph and Hamsa are used to protect CNN.com and Google.com, the allergy attacks we've proposed can cause 20% to 100% of HTTP requests to be dropped.

We conclude this chapter by arguing that the root of the vulnerability against allergy attacks lies at the semantic-free signature generation process used in many network-based IPSs. By semantic-free, we mean signatures are generated without considering how the properties matched by the signatures contribute to successful attacks (i.e. citing the authors of [65], input to the signature generation are treating as "opaque "bag of bits"""). In other word, the different components of an attack (e.g protocol frame for control hijacking, filler bytes for buffer overflow, return address used to direct control to worm payload, or the worm payload itself) are treated the same in a semantic-free signature generation process. This property makes it possible for the vulnerable IPSs to confuse part of the targeted traffic as an invariant property of an "attack", and use it as an "attack signature", and thus is a precondition for successful allergy attacks.

While a semantic-free signature generation appears to be the only kind of analysis possible for an IPS that only observes network traffic, we note that IPSs that collect information on attacked hosts are not automatically immune to allergy attack. In the following, we will give two examples of IPSs that employ some host-based information for signature generation and are still vulnerable to allergy attacks. We believe these examples will highlight the importance of collecting the "right" attack information for the analysis in an IPS, and provide a good starting point for our discussion on building IPSs that collect and analyze host-based attack information.

2.6.1 Attacking TaintCheck

TaintCheck [66] is a novel intrusion detection system that uses dynamic taint analysis to keep track of tainted data, i.e. data that originates or is derived arithmetically from an untrusted input. An alert is generated whenever the tainted data are used in an unsafe way, e.g. used as a jump address. Furthermore, TaintCheck can obtain the value of tainted data that is used for unsafe operations. In an injected code attack, this will mean the value used to overwrite a function pointer or return address. In [66], Newsome et al suggest using the most significant three bytes of this value as a signature for attacks exploiting the same vulnerability. The evaluation in [66] shows that this preliminary signature generation scheme is very effective in detecting attacks and results in low false positives. However, such signature generation process is vulnerable to allergy attacks; in order to block out the target request, the attacker simply modifies a real control-hijacking attack to overwrite the corresponding function pointer or return address with a 3-byte subsequence in the target request.

2.6.2 Attacking FLIPS

FLIPS [56] is a system that generates signatures to filter HTTP requests. It uses a network-based anomaly detection system called PAYL to identify suspicious packets. PAYL detects anomalous packets by using a normal profile that describes the byte-frequency distributions for normal traffic of different length and destination port. Any packet which shows significant

deviation from the profile will be labeled suspicious. In FLIPS, all suspicious requests are cached. FLIPS also employs a host-based intrusion detection system called instruction set randomization (ISR). In addition to detecting attacks with zero false positive, the ISR also identifies the beginning of the attack payload. When ISR detects an attack, FLIPS will copy the first 1KB of the attack payload. The memory copied will be matched against the cached suspicious request based on a similarity score computed as 2C/(S1+S2), where C is the longest common substring between the packet and the captured payload, S1 and S2 are the length of the two string being compared. The request most similar to the payload will be used as the signature of the worm. Any incoming request that is sufficiently similar to the signature will then be dropped.

Now let us consider an attack targeting HTTP requests to the homepage of UTCS (which is 475-byte long). The attack involves sending two attack packets at the same time. The first packet is the one to be used as the worm signature at the end. This packet is constructed by appending to the target request a byte sequence that we call "gibberish". The gibberish is intended to make the packet suspicious to PAYL. In our attack, we will have a 100-byte gibberish, all filled with a byte that rarely occurs in normal HTTP requests. This should make the packet sufficiently anomalous to be cached by FLIPS. The second packet contains a real code injection attack (e.g. the one from Code-Red), with the content of the first packet appearing at where payload should be placed (this makes the second packet around 1050 byte long). Once this second packet triggers the alarm from ISR, the cache will be

searched for the suspicious request responsible. With similarity computed as described above, the request from the shorter first packet will achieve a higher similarity score of 0.73 (when compared to the 1KB "worm payload" identified by the ISR, which contains most of the 575 bytes in the first attack packet, and whatever follows in the memory when the attack is detected). Thus this first request will be used as the signature, and filter all instances of the target requests in future traffic (the similarity score between the target request and the signature is 0.90, which is much higher than the threshold used in [56]).

One final point that is worth nothing about the allergy attacks against both FLIPS and the IPS based on TaintCheck is that, even though both of them use a zero false-positive mechanism to detect attacks, neither mechanisms provides any information about the state of the attacked process before the control hijacking occurs. In other word, the signature generation process of FLIPS and the IPS in [66] only rely on information about what state the attacked process is in either at the point of control hijacking, or after the control has been hijacked. As we can see in the above attacks, attackers have almost complete control over such states; i.e. they can easily manipulate what the IPS "sees", and this proves to be a problem. Thus, we believe information about how the attacked process reaches the final compromised state is much more valuable than information regarding what state the victim process is in after the hijacking has occurred (and detected). We will elaborate on this point in the next chapter, and show why this turns out to be a difficult problem facing IPS design.

Chapter 3

The LAIDS/LIDS Framework for Building Host-based IPSs

In the previous chapter, we've presented the difficulties facing network-based IPSs and highlighted the need to utilize information collected from the victim hosts in generating countermeasures against detected attacks. As we will show in this chapter, finding the right mechanism to collect useful attack information is a non-trivial problem. This is true even in the traditional setting where honeypots can be used for heavy-weighted monitoring/analysis.

In the following, we will refer to any IPS that employs information collected from attacked systems "host-based IPSs". We will start our discussion by presenting related work in this area. After that, we will introduce a class of intrusion detection systems called the Lazy-able Intrusion Detection Systems (LAIDS), which is very suitable for collecting attack information in a host-based IPS. We will then present a framework for building IPSs based on LAIDS; we call this framework the LAIDS/LIDS framework. Under this framework, once the LAIDS detects a new attack, the generation of countermeasure against the attack is a mere retrieval of a constant amount of data collected. We note that the LAIDS/LIDS framework is designed for the tra-

ditional settings where honeypots are used to collect attack information, and thus faces the same problems as traditional host-based IPSs (as described in Chapter 1). Nonetheless, we find that our collaborative intrusion prevention framework to be largely inspired by ideas in the LAIDS/LIDS framework.

3.1 Related Work

A study of the literature in host-based IPSs shows that a three-phase, detect-analyze-protect framework is employed by almost every existing system, with the LAIDS/LIDS framework and the CIP framework being no exceptions. In the following, we will briefly describe this three-phase framework, and show how each of the three phases is implemented in existing systems.

3.1.1 The Framework

- In the detection phase, attacks against protected systems are captured in the wild.
- Based on the data collected from the first phase, the analysis phase generates countermeasures against the observed attack/vulnerability.
- Finally, in the protection phase, the countermeasures generated will be distributed and applied, so that all protected hosts will be immuned to the detected attacks.

3.1.2 Detection

The major challenge in the detection phase is to collect information to facilitate the countermeasure generation process that follows the detection of a new attack. A wide range of techniques have been applied in the detection phase. For example, the earlier system in [76] employs a GCC extension called ProPolice that protects programs from stack-smashing attacks (in a manner similar to StackGuard). On the other hand, [66, 63] employed dynamic taint analysis to detect the attack and to record how the targeted process reaches the illegal state that leads to the detection. [7] assumes a full execution trace available for the analysis phase (though it is not clear how this is achieved in [7]). A point worth nothing is that both the dynamic taint analysis in [66, 63] and the execution tracing in [7] require a close to full emulation of the monitored process. Other host-based IPSs employ more typical mechanisms in the detection phase (e.g. ASLR in [97, 54], data non-executable technology in [15]). A distinguishing property of the lighter-weight detection mechanisms like ASLR, data non-executable and ProPolice is that they do not provide any non-trivial information about the state of the attacked process before the actual control hijacking occurs. As we will see, this means the analysis process in IPSs that employ these detection mechanisms are usually complicated, risky and possibly inaccurate.

3.1.2.1 Analysis

The analysis techniques in host-based IPSs generally fall into two categories, depending on whether the analysis utilizes information about how the target process behaves before the control hijacking occurs. First, let's consider IPSs that only analyze the state of the target process AFTER the attacker takes over the control.

3.1.2.2 Analysis with "Postmortem" State Only

Analysis performed using only information collected after control has been hijacked are usually very similar to FLIPS and TaintCheck discussed in Sect. 2.6. For example, [97] is similar to TaintCheck and tries to reverse engineer both the value used for overwriting the targeted control structure, as well as the address of this data structure. With these critical values identified, signatures can be directly generated to filter out any network packets which contain these values. However, [97] can be defeated by effectively the same allergy attack as presented in Sect. 2.6. In general, systems that attempt to generate signatures based on "critical values" employed in the process of control hijacking are also very susceptible to evasion by attack polymorphism. This is because, as shown in [19], attackers can have a lot of choices for these "critical values", and thus easily bypass any signature generated by using a different value.

An alternative method of utilizing the critical values recovered from the hijacked process' address space is to use them to identify the attack packets

among all recently received packets. FLIPS presented in Sect. 2.6 is an example of IPSs that follow this approach, another example is [54]. In [54], after the attack packet is found, the IPS will try to find out what type of message under the corresponding application protocol the identified packet belongs to, and what is anomalous about that packet (e.g. certain field being too long, or have a rare value), and use these information in signature generation. However, it is questionable whether the attack packets can be correctly identified by using the extracted critical values as search keys. It appears entirely possible that [54] is vulnerable against the same form of attack against FLIPS as presented in Sect. 2.6; in particular, [54] can be misled to use the wrong attack packet for signature generation if the attacker can send two packets in parallel, the first being the real attack, the second contains the byte sequence that will be used as key during the search for attack packet, but appears more like an attack than the first packet. Once the attacker tricks the IPS into generating signature based on the "attack packet" of his choosing, both evasion and allergy attack are possible. Evasions based on techniques similar to [70, 65] is also possible against [54]; even if the IPS successfully identify the actual attack packets, attackers can introduce into these packets "spurious " fields that are anomalously long or have byte frequency distribution that deviates from the norm more than the field that should be used in the attack signature.

Another approach taken by host-based IPSs that employ a simple/typical detection mechanism is to use any alert to trigger an attack replay in a closely monitored honeypot. Under this approach, the IPS will keep a buffer of re-

cently received packets. Once an attack is detected, all the buffered packets will be sent to, and processed by a highly instrumented honeypot, with the hope of reproducing the detected attack and collecting detailed information about how the malicious packets are processed. Examples of systems that use this approach include [76, 15]. However, we note that it may not always be easy to reproduce a detected attack using logged packets alone. First of all, if the attack involves multiple packets, attackers can try to evade the IPS by sending many irrelevant packets between the first and the last attack packet, so the IPS will not have enough buffer space to hold all the attack packets. To make this problem of insufficient buffer even worse is the observation in [19]: some attacks involve attack packets being processed by multiple processes. Thus merely replaying buffered input to the compromised process, or from the attacking IP address[1] is insufficient.

3.1.2.3 Analysis with Information Collected Before Hijacking

As for IPSs that employ information about the state of the attacked process before the occurrence of control hijacking, there are two typical approaches for analysis, each generating a different kind of countermeasures. The first approach outputs "execution filters", which contain information that allows protected systems to detect future instances of attacks for which the countermeasure is generated. In fact, IPSs based on the LAIDS/LIDS framework and the proposed CIP framework belong to this category. Another example of

[1] [15] used this second approach

systems that output execution filters is [63]. In [63], dynamic taint analysis is used to detect attacks and record the flow of tainted data. Once an attack is detected, an execution filter will be generated to identify all the instructions that need to be monitored / traced for the attack to be detected. In particular, the instruction at which the attack is detected is identified, so protected systems can check for unsafe use of tainted data at this instruction. All the instructions that are involved in propagating tainted data from the malicious input to the contaminated control data structure used at the point of detection are also identified. If the protected systems record the flow of tainted data at all these instructions, any future instances of the attack will result in the same control data structure being tainted, and thus can be detected at the same point.

We note that the idea behind [63] is very similar to the LAIDS/LIDS framework. However, there are two subtle differences. First of all, as we will see, the LAIDS/LIDS framework has a much lighter weight analysis, and outputs much simpler countermeasures. In particular, the analysis under the LAIDS/LIDS framework requires the identification, and monitoring at one instruction only, this means both the analysis and the monitoring performed at protected systems are many times simpler than that in [63] (which requires the monitoring/tracking of 200 instructions to detect SQLSlammer). Furthermore, in [63], the monitoring performed on the honeypots (which is used for detecting new attacks) is slightly different from that on the protected systems (for stopping known instances of attacks). As a result, the zero false positive

rate of the original dynamic taint analysis no longer holds on the protected systems, since instructions that may "untaint" some tainted data may not be monitored, and false positives may arise when these untained values are used at instructions where checking for unsafe use of tainted data are performed. Thus the monitoring mechanism running on protected systems has to be modified in order to handle these false positives. On the other hand, the LAIDS/LIDS framework uses the same monitoring technique on both the honeypot and the protected hosts; as a result, the mechanisms at both sites will have exactly the same properties.

Other examples of host-based IPSs that generate execution filters include [76, 78]. In [76], once ProPolice detects a buffer overflow and identify both the vulnerable function and the buffer involved in the attack, the vulnerable function will be modified so that the buffer that got overflown is allocated on the heap; as a result, any overflow attempt will not be able to hijack the control by corrupting the return address on the stack. However, the system presented in [76] can only handle stack buffer overflow, while our prototype IPS based on the LAIDS/LIDS framework can detect any injected code attack. Furthermore, the work in [76] can be seen as only an isolated IPS, and provides no general direction for identifying a mechanism for detecting attack or any general method to generate countermeasures once an attack is detected.

Similarly, [78] proposed to stop attacks on protected systems with the "selective transactional emulation" (STEM) system, which allows any chosen piece of code to be executed under close monitor (to allow detection of attacks).

STEM also makes it possible to "roll back" any modification to the CPU and memory state made by the monitored code if an attack is detected during the emulation. With this mechanism, countermeasures against attacks will only need to identify the code that needs to be emulated. Once again, the work in [78] can be seen as very similar to the LAIDS/LIDS framework. However, the system in [78] is highly restricted to stack buffer overflow, and did not provide any apparent method of extending it to cover other kinds of attacks. Another point that is worth noting about [78] is that it is one of the very few work that proposed a method for recovering from any attacks detected by an execution filter. The recovery mechanism proposed in [78] is called "forced return recovery". Under the forced return recovery, if the STEM is started at a function that is allowed to return with failure, an attack detected during the STEM emulation can be gracefully handled by rolling back the CPU and memory state to the beginning of the emulation, and return from the emulated function with a value that signifies failure. However, we note that this solution only works if the emulated code does not modify any persistent state of the protected system (i.e. it did not write to disk, send/receive network traffic, etc). Otherwise, the roll back provided by STEM and the forced return may leave the protected system in an inconsistent state. We believe a recovery mechanism that allows users/software vendor/system administrator to specify the recovery policy for various kinds of resource is necessary for reliably recovering from any attacks. A similar idea has been implemented in [55]; nonetheless, the work in [55] appears to focus mostly on memory state and

network traffic, without paying much attention to the file system. Another potential problem that is acknowledged by the authors is that repair policies in [55] can be hard to specify. Finally, we note that in this dissertation, we only focus on generating execution filter that allows attacks to be detected as early as possible and prevents them from causing damage to the protected systems; we'll leave the development/incorporation of techniques to maintain normal functioning of the attacked process/system to future work.

The second approach for performing analysis using information about how the malicious inputs are handled by the vulnerable process tries to output "programs" to filter out all future instances of the detected attacks. The main idea behind this approach is to extract the weakest conditions in the incoming packets/inputs that are necessary for reproducing the same/similar control/data flow that leads to the attack's detection. For example, in [15], the analysis involves the replaying of attack packets in a sandbox environment that performs both dynamic taint analysis and execution tracing. The taint analysis maintains for each tainted memory location, register and CPU flag a formula that describes how the tainted value is generated from the malicious input. For the execution tracing, at each conditional control transfers/move and set instructions that involves a tainted value, a "filter condition" will be added to the output "filter program" being generated, so that the inputs identified by the filter program will result in the same control path by setting the same condition flags as during the attack replay. The final filter program is then a program that computes the conjunction of all these conditions. Note

that the analysis performed by [15] is largely linear to the number of instructions executed before the detection of the attack, during the processing of the malicious input, and such analysis is performed under effectively full emulation of the actual processing; thus, the time required for the analysis can be very long. In fact, the experiments in [15] reported that the time required for the entire analysis process can be in the order of seconds (which is very long for modern computers).

As another example, [7] generates three different kinds of "programs" to identify attack traffic, the authors of [7] called these programs "turing machine signature", "symbolic constraint signature" and "regular expression signature". The "turing machine signature" is basically a record of all the instructions executed during the processing of the malicious input, until the point that leads to the attack being detected. If a tested input leads the turing machine through the path taken by the detected attack, the conditions that allowed the attack to be detected at the first place (e.g. unsafe use of tainted data) will be checked, and the input will be considered as an attack if the check is successful. Otherwise, if the check fail, or the input leads the turing machine through a different control flow path, the input will be considered benign. The generation of the "symbolic constraint signature" involves the symbolic execution of the turing machine signature. We can consider the symbolic constraint signature very similar to the program output by [15]. Finally, the generation of the "regular expression signature" involves the solving of the constraints in the "symbolic constraint signature", and the authors in

[7] proposed some methods to simplify the task, then employ a model checker to solve the simplified tasks.

One major emphasis of the work in [7] is the computation of vulnerability-specific signatures. In particular, instead of only computing signatures for the control flow path taken by the detected attack, [7] performs binary analysis to identify all control flow paths that lead to the point where the attack is first detected. Both the generation of the turing machine signature and the derivation of the regular expression signature in this case is basically the same as the single path case, except that the derivation of the symbolic constraint signature involves static analysis technique to handle loops in the turing machine signature. However, there are two points we would like to point out. First of all, the proposed signature generation in the multi-path scenario is far from practical; binary analysis techniques on x86 systems should still be considered unreliable, especially on Windows systems (see Chapter 5 for one of the many reasons why this is the case). Furthermore, as the experiments in [7] indicates, the signature generation in the multi-path scenario either cannot scale up to real-life software libraries, or is extremely slow even for a very simple test case. A more serious issue with the multi-path signature generation proposed in [7] is that, it does not cover all variants of attacks that exploit the vulnerability under consideration. In particular, it is entirely possible for attack variants to avoid the execution point where the attack is detected on the honeypot. This is especially true for vulnerabilities that allow writing arbitrary value to arbitrary addresses (e.g. heap overflow). In this case, the attackers can choose

to contaminate different function pointers. Furthermore, even if the attacker always targets the same control structure, the contaminated data may be used at many different execution points.

As a conclusion, we found that IPSs that only analyze information collected after an attack is detected are usually vulnerable to allergy attacks and worm polymorphism. On the other hand, IPSs that analyze information concerning how the victim process reaches the point of control hijacking and generate some signature or input filter to stop the detected attack tend to have very complicated (and thus slow) analysis process. For IPSs that output execution filters, while they have simpler analysis process, some of them are restricted to very specific class of attacks (e.g. [76, 78]). Furthermore, we find that guidelines for finding the right detection mechanism for this class of IPS is lacking, and the result of using unsuitable components in the detection phase is usually more complicated analysis process, and/or less reliable, harder to deploy execution filters.

3.1.3 Protection

As compared to the first two phases in the detect-analyze-protect framework, work in the protection phase is relatively straightforward. For IPSs that output signatures / filters to identify malicious traffic / inputs, the countermeasures are simply deployed either at network perimeter defense, or at proxies to the protected servers / applications, and the performance overhead of signature matching or input filtering is usually very low.

On the other hand, if the IPS outputs execution filters, these filters have to be distributed and applied to all protected hosts. In the simplest case (e.g. [63, 78]), the protected hosts will be running some light-weighted IDS that is capable of stopping the detected attack when guided by the information in the execution filters. The performance overhead incurred on the protected hosts will then depend on how light-weight this IDS is. For example, in [63], each execution filter output by the taint-analysis-based detection/analysis process will require the protected hosts to keep track of how tainted data propagates from the attack packet to the control data structure targeted for the control hijacking. This means multiple data movement instructions will have to be monitored in order to defend against one single attack. Furthermore, we believe at least some of these monitored instructions will appear in some loop that is executed every time an incoming packet is processed. Thus, the monitoring of these instructions may lead to a non-trivial overhead (in [63], a 3% overhead is reported when a single execution filter is applied). Similarly, each execution filter in [78] will require the emulation of at least one function in the protected process. This will once again imply a non-trivial performance overhead for each filter applied. On the other hand, in [76], the protected systems have to be restarted to use a "hardened" binary image generated based on the information in the execution filter.

3.2 The LAIDS/LIDS Framework

In the following, we will give the definition of a special class of IDS that we call the Lazy-Able Intrusion Detection Systems, and explain what makes LAIDS so suitable for collecting attack information in an IPS that outputs execution filter. Based on the definition of LAIDS, we will define a class of IDSs that is derived from the LAIDS, we call this class the Lazy Intrusion Detection Systems (LIDS). After that, we will present the LAIDS/LIDS framework for building IPSs using the two related classes of IDSs. The main advantage of the proposed framework is that it has a very simple process for countermeasure generation. Furthermore, even though not many existing IDSs fit our definition of LAIDS, we believe our work on the LAIDS/LIDS framework will provide useful guidance to help design IDSs that are suitable for the detection phase in an IPS. One advantage of using our definition of LAIDS as an IDS design principle is that once the IDS is implemented and properly evaluated, the properties of the IPS that result from plugging the IDS into the LAIDS/LIDS framework is entirely known to us. This is because both the false positive and false negative properties of the countermeasures generated by such IPS are completely inherited from the underlying LAIDS.

3.2.1 Defining LAIDS and LIDS

The main idea behind the LAIDS/LIDS framework is to simplify the analysis process in the IPS by using the same mechanism to detect new attacks on the honeypot and to stop known attacks on protected hosts. However, in

order to maintain the performance overhead at a very low level, we will be running a down-tuned version of the detection mechanism on the protected hosts, and the execution filter generated by the IPS will be responsible for configuring the IDS on the protected hosts so it will have high detection rate against known attacks. As such, we can defend protected hosts against attacks previously seen on the honeypots at a very low cost.

It turns out that this goal of performing a down-tuned intrusion detection on protected host while maintaining very high detection rate against known attacks is possible only in a restricted class of IDSs that we call the "Lazy-Able" Intrusion Detection Systems (LAIDS). In particular, we define the LAIDS as follow:

> LAIDS is the class of host-based anomaly detection systems which perform self-contained analysis at every inspection point.

Where inspection points refer to occasions at which we perform analysis for intrusion detection (under whatever monitoring mechanism employed by the LAIDS concerned), and self-contained analysis means the analysis relies only on information specific to the inspection point (i.e. those maintained by the underlying OS or runtime environment), or information collected from a small, constant number of places in the program's execution. Note that this requirement implies the analysis at different inspection points should be independent. Otherwise, the analysis at one point will implicitly depend on the information collected from analysis at some other points, and consequently,

the monitoring of one point will lead to the inclusion of many other inspection points, which will adversely affect our ability to tune down the performance overhead on protected hosts. For example, both the dynamic taint analysis in [63] and the traditional system-call-based IDSs cannot be counted as LAIDS. In the first case, the analysis at any control transfer instruction may rely on information collected from a large number of points that are responsible for propagating tainted data to the memory location/register under consideration. In the latter case, the intrusion detection performed at each inspection point usually depends on the history of all system calls made by the monitored process/thread.

Our framework also needs a monitoring mechanism that is configurable and performs monitoring only at points where it is necessary. We want to avoid any mechanism that incurs a constant base performance overhead regardless of the number of inspection points being monitored. For example, the monitoring of the popular system-call interface is not suitable for our purpose. However, we do not include this requirement in our definition of LAIDS because in most cases, it is easily achievable through the use of software breakpoints, as we will show in Sect. 3.3.

With these restrictions, we can scale down the LAIDS into a light-weight IDS, and we call this the Lazy IDS (LIDS). As mentioned before, the LIDS performs the same analysis as the corresponding LAIDS. However, intrusion detection will only be performed at a configurable subset of inspection points on the LIDS. Thanks to the self-contained analysis, the effectiveness of

intrusion detection at any set of inspection points will be the same for both the LIDS and the LAIDS. The LIDS will only miss an intrusion detected by the LAIDS if the corresponding inspection point is not used. This is where the countermeasures generated by the IPS come into play; they are responsible for identifying the inspection points that need to be monitored so that attacks detected by the LAIDS on the honeypot will be stopped by the LIDS on protected systems.

3.2.2 Putting It All Together

After defining LAIDS and LIDS, we now summarize how to use them to construct IPSs. In the LAIDS/LIDS framework, the detection and analysis are performed on a machine running the LAIDS (the honeypot), while each client system runs the LIDS. Now we will see how LAIDS/LIDS based IPSs implement the detect-analyze-protect framework:

- In the detection phase, once the LAIDS detects an intrusion, the inspection point at which the attack is detected will be identified. This identity is all that the LIDS on client sites need as a countermeasure.

- Though the countermeasures from the detection phase is usable on client sites, they can be refined in many aspects in the analysis phase. For example, the inspection points can be represented in a more portable form, and information that tells what version of software the countermeasure applies to can be included. Finally, it may be necessary to add infor-

mation that allows clients to verify the authenticity or validity of the countermeasures.

- After the analysis phase, countermeasures will be distributed to different client sites. Upon receiving countermeasures, the LIDSs reconfigure themselves to protect client systems against the new attacks by including the inspection points identified in the list of monitored points. Countermeasures generated by untrusted sites may also be validated at the client systems before they are applied.

After seeing how countermeasures are generated and applied in the proposed framework, we analyze their effectiveness. We shall assume that both the honeypot and the client system run the same software. Our analysis will consider two types of attacks: previously detected attacks, and attacks against a previously exploited vulnerability. Analysis specific to our prototype will be presented in Sect. 3.3.

The effectiveness of the countermeasures against previously detected attacks should be obvious. Based on our assumption, both the detector and the client will process the attack in exactly the same manner. Since the attack is detected by the LAIDS before, the corresponding inspection point will be monitored in the LIDS, and leads to positive detection.

As for the different exploitations of the same vulnerability, let us consider an attack as a two stage process of control hijacking and payload execution (existing code executed on behalf of the attacker is also considered

as a payload). We will argue below that the effectiveness of the countermeasure in this case is determined by whether the detection which leads to the countermeasure generation occurs in the first stage or the second stage of the attack.

The activities in the first stage are mostly vulnerability specific. There is very limited freedom in how an attacker can exploit a particular vulnerability. In other word, the processing of various attacks against the same vulnerability will be mostly the same in this stage. Countermeasures generated based on the attacked system's behavior in the first stage will therefore be effective against all these attacks. On the contrary, there are a great variety of payloads that the attackers can execute in the second stage. Polymorphism / metamorphism of attack payloads will further increase this variety. Since not all attacks against the same vulnerability will behave the same in the second stage, countermeasures based on behavior in this stage may not be effective against new exploits of a known vulnerability (similar observations are found in [18]).

Finally, we should point out that LAIDS and LIDS (or any IPS that relies on execution filters) cannot completely prevent client systems from processing malicious inputs; they can only stop the processing at some point. As a result, the client systems may have reached some unsafe state when attacks are detected, and a recovery mechanism is necessary. Nonetheless, a LIDS that stops attacks before payload execution will greatly simplify the system recovery. The LIDS in our prototype IPS is an example in this category.

3.3 A prototype LAID/LIDS based IPS: Lazy Shepherding

After describing our LAIDS/LIDS framework, we will present our prototype LAIDS/LIDS based IPS. The core of our prototype uses Program Shepherding as proposed in [43] as the LAIDS.

3.3.1 Program Shepherding: the LAIDS

Program Shepherding is a protection system that monitors control transfer instructions to avoid jumps to malicious code. By enforcing different security policies at all control transfers, Program Shepherding can detect attacks ranging from injected code attacks to existing code attacks. A property that makes Program Shepherding an excellent LAIDS is that intrusion detection at a control transfer relies on information available at that point only, and does not depend on the analysis at any other control transfers. Nonetheless, the implementation in [43] cannot protect system processes, which are not started by Program Shepherding (such as svchost.exe and lsass.exe in Windows). In order to overcome this shortcoming, we choose to implement our own Program Shepherding, instead of improving on the existing one. We emphasize that our implementation is only intended to demonstrate the usefulness of the LAIDS/LIDS framework. The quality of this implementation cannot be compared with the careful engineering presented in [43], nor is it our intention to do so.

We implement our Program Shepherding on Windows, and base it on

the "single-step-on-branch" feature of the Intel IA32 architecture, which is available in all P6 and later family processors. This feature generates an interrupt at every control transfer, and allows us to perform the necessary intrusion detection at the interrupt handler. For our implementation, we enforce two simple security policies:

- non-execution of data, and
- return statements should pass control to an instruction following the call-site

In our implementation, we detect data execution by using the Virtual Address Descriptor (VAD) tree maintained by Windows, which allows efficient look up of the protection attribute of any memory region in the address space, and we consider any memory region that is writable but not copy-on-write as data. As for the restriction on the destination of return statements, we can enforce it by scanning the code that precedes the target of the return statement in question. The return is valid only if a call instruction is found immediately before the return target.

Once our Program Shepherding detects an attack, a countermeasure will be generated based on the offending control instruction. Instead of using its virtual address, we will identify the instruction with its containing module and its offset within the module. This guarantees the instruction will be correctly located, even if the module is dynamically loaded at a different base address.

3.3.2 LIDS in Lazy Shepherding

In Lazy Shepherding, the LIDS is a light-weighted Programming Shepherding configured to enforce our security policies at a small set of control transfer instructions identified by the LAIDS. The LIDS starts by loading all DLLs for which inspection points are to be inserted. For each DLL involved, the LIDS will temporarily remove its write-protection to prevent a copy-on-write. The inspection points will then be inserted by modifying the binary image of the DLL. This will make the insertions visible to all processes, both existing ones and those created afterwards. After inserting the inspection points, the LIDS simply waits until the execution reaches some inspection point, and performs the necessary intrusion detection, where the intrusion detection is implemented the same way as in the LAIDS. In the following, we will give more detail on how we insert inspection points and how intrusion detection is performed.

Inspection points are inserted by replacing the first byte of the corresponding instruction with 0xcc. This turns the targeted control transfer instruction into a "trap-to-debugger" instruction, the execution of which will generate a breakpoint exception. In the exception handler, we will perform the necessary checking as well as emulate the original control transfer instruction. By modifying in-memory images, our approach can insert/remove inspection points without restarting the machine. It also avoids the problem of writing to Windows system files (like ntdll.dll) which are heavily protected.

Our current implementation of LIDS does not insert inspection points

Vulnerability ID	Exploited by	Exploit type	LAIDS?	LIDS?	Monitored instruction
VU#484891	SQL Slammer	SBO	Yes	Yes	ret
VU#568148	MSBlast, Welchia	SBO	Yes	Yes	ret 0x8
VU#753212	Sasser	SBO	Yes	Yes	ret 0xc
VU#806278	N/A	HBO	Yes	Yes	call [edi+0x74]
VU#228028	N/A	SBO	Yes	Yes	ret 0xc
VU#116182	N/A	HBO	Yes	Yes	call ecx
VU#625856	N/A	SBO	Yes	Yes	ret 0x18
VU#842160	N/A	HBO	Yes	Yes	call [ecx]
VU#939605	N/A	HBO	Yes	Yes	call [ecx+0x8]

Table 3.1: In this table, "Vulnerability ID" refers to the US CERT ID of the vulnerability. For "exploit type", SBO and HBO refers to stack-based and heap-based buffer overflow respectively. "LAIDS?" and "LIDS?" indicates whether the attack is detected by the LAIDS and the LIDS (with countermeasure applied) respectively. "Monitored instruction" refers to the instruction at which the attack is detected on both LAIDS and LIDS.

into executables. Nonetheless, our experience shows that most vulnerabilities are found in the DLLs, instead of executables. Furthermore, it should be very straightforward to extend our implementation to protect ".exe" files.

3.3.3 Evaluating Lazy Shepherding

3.3.3.1 Effectiveness of Countermeasures

We have tested Lazy Shepherding against attacks on 9 Windows vulnerabilities, including the very "successful" MSBlast, Welchia, Sasser and SQL-Slammer. Details about the tested attacks, as well as the results of our experiments are given in Table 3.1.

As shown in Table 3.1, our implementation of Program Shepherding detects all the tested attacks and generates effective countermeasures for them. Our experiments also indicate that the countermeasures generated are mostly vulnerability-specific.

In all the 5 stack-based buffer overflows tested, the return instruction that starts payload execution is identified (and used in the corresponding countermeasure). Since this critical point in control hijacking is fixed for the vulnerability, countermeasures generated for stack-based buffer overflows are vulnerability-specific. They will be effective against all attempts of the identified vulnerabilities.

On the other hand, for vulnerabilities like heap buffer overflows that can overwrite arbitrary function pointers, Lazy Shepherding cannot detect the control transfer based on the dirty function pointer. We can only detect the execution of the register springs (register indirect jump/call instructions) where the modified function pointers point to. Since the attackers can choose different register springs to transfer control to the injected payload, the countermeasures thus generated is not entirely vulnerability specific. Nonetheless, as pointed out in [18], register spring instructions are very rare, thus they only allow very limited polymorphism, which can be handled by a small number of inspection points in the LIDS.

3.3.3.2 Performance overhead incurred by LIDS

We measured the performance overhead incurred by our LIDS with four SPEC2000 benchmarks, bzip2, gzip, gap4ra and link41a. We measured the overhead incurred when 100, 200, 300, 400 and 500 artificial "countermeasures" are applied to the LIDS respectively. For each benchmark and each of the 5 data points, we generate 10 random sets of "inspection points" (i.e. 10 sets with 100 "countermeasures", 10 sets with 200, etc). These sets are generated by randomly picking control-transfer instructions among the DLLs imported by the benchmark tested.

We obtain an average overhead caused by each set by running the benchmark with the countermeasures applied 10 times. The 10 average overheads that correspond to the same number of countermeasures applied will be averaged. The results of our experiment are presented in Fig. 3.1.

As shown in Fig. 3.1, the performance overhead incurred by LIDS is less than 3% for all the four benchmarks tested, even when 500 "countermeasures" are applied. Considering the fact that only around 1600 vulnerability notes are published by US-CERT starting from Sept 2000, and that countermeasures generated by our scheme are mostly vulnerability-specific, we believe 500 is quite a significant number. Furthermore, we argue that experimenting with randomly picked inspection points is very realistic: occurrence of inspection points in real systems is expected to be at least independent of how often that program point is reached. In fact, it is possible to have a negative correlation between the two: vulnerabilities are more likely to appear in rarely executed

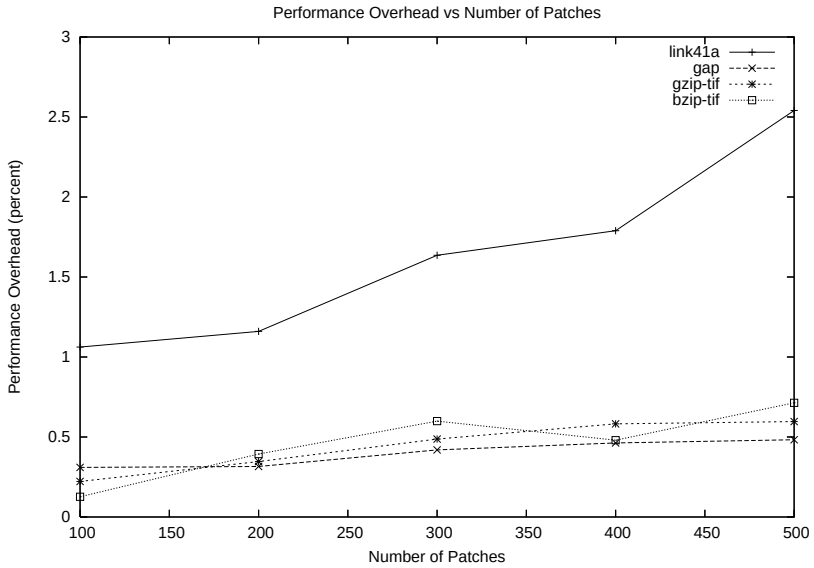

Figure 3.1: Performance overhead incurred by LIDS when different number of countermeasures are applied.

code.

Finally, we would like to point out that our experiments do not measure the overhead caused by inspection points in executable files. However, we believe the above figures are still realistic prediction of the performance impact caused by the LIDS. First of all, we expect many of the inspection points inserted in an executable file will only be executed at a very low frequency under normal circumstances (i.e. when the system is not being attacked), based on an argument similar to the above. More importantly, the performance impact caused by countermeasures in a DLL is much more widespreaded: it is experienced by all processes sharing the library, while countermeasures in an executable only affect a single process.

Chapter 4

Random Inspection-Based IDS

We will now turn our attention to a class of LAIDS that is particularly suitable for our collaborative intrusion prevention framework; we call this the Random Inspection-Based Intrusion Detection Systems. In this chapter, we will present the general idea of random-inspection-based IDS, followed by the details of how we implement a prototype IDS based on this idea, and evaluate the detection rate and performance overhead of this prototype. In the next chapter, we'll provide a detailed study of the false positive behavior of our prototype. Finally, we will demonstrate how this class of IDSs fit into the LAIDS/LIDS framework and how they can be used to implement the idea of collaborative intrusion prevention in Chapter 6. As in all the previous chapters, we'll start our discussion with a brief introduction of the related work.

4.1 Related Work

There are three general approaches for host-based intrusion detection, namely misuse detection, anomaly detection and specification-based intrusion detection. In the following, we will briefly go over the work done in each area.

4.1.1 Misuse Detection

Misuse detection techniques detect intrusion by matching incoming traffic/files against signatures of known attacks. The most common examples of misuse detection systems are the various antivirus software available. However, it is a common consensus in the research community that misuse detection systems are inherently vulnerable to attack polymorphism (since signatures are mostly designed to match attack payload, which can be easily polymorph), thus very little research has been done in the area of misuse detection. Recently, the industry is also starting to recognize the difficulties facing misuse detection.

4.1.2 Anomaly Detection

The idea of anomaly detection is to build a model of normal system behavior and check observed behavior against the model; any behavior that significantly deviate from the model will then be marked as an attack. The first anomaly detection system that we are aware of is proposed in [20] in the 1980's. At that time, the only known mechanism for monitoring the behavior of processes is the audit-log. The kernel and other system components are responsible for monitoring process behavior and make this result available in audit-logs. The IDS will then read the audit-log and determine whether an intrusion is observed based on what is read. A new monitoring mechanism only came on the scene when [24, 34] proposed system-call-based anomaly detection. By using system-call traces for intrusion detection, an alternative monitoring

mechanism, namely the monitoring of the system-call interface is implicitly introduced. Another major contribution of [24, 34] is the introduction of black-box-profiling technique. This is a technique that allows the normal behavior of a process to be profiled by just observing its normal execution. The process is treated like a black box and the availability of the underlying code being executed is not necessary. The normal profile thus generated can then be used to match against observed behavior for intrusion detection.

Due to the richness and timeliness of the information available at the system-call interface, system-call-based anomaly detection has become a mainstream approach in intrusion detection. A lot of work has been done in enhancing system-call-based detection [44, 48, 47, 40, 52, 72, 93]; most of them focus on the profiling technique. At the same time, black-box profiling for the traditional audit-log monitoring mechanism has also received a lot of attention [10, 16, 28, 51, 94].

An alternative approach for system-call-based anomaly detection called "behavioral distance" has also been proposed in [25, 86, 26]. In this new approach, a form of "n-version programming" is employed. Instead of building a model of the system call sequence issued by a single process, this new approach tries to come up with a metric ("distance") for comparing the system call sequences made by different implementations of the same software (e.g. web server). The idea behind this approach is that the different implementations under observation should process the same benign input in a similar manner; on the other hand, since attacks are usually implementation-specific,

the target process should process the attack in a manner quite different from the others. Thus, if the behavior of one observed process shows significant difference from the others according to the distance metric, it is considered to be under attack.

Despite all the work done in enhancing both system-call-based and audit-log based anomaly detection, the underlying monitoring mechanisms have remained largely the same. Monitoring at the system-call interface and monitoring through the system audit log facility are still the two mainstream monitoring mechanisms. There are some other monitoring mechanisms proposed (implicitly with the use of new observable behavior for anomaly detection, such as [8, 35, 95]), but none of these is as general as the two traditional approaches.

The success of system-call based anomaly detection also brings a lot of studies [84, 83, 89] that try to find out the limitations and weaknesses of these system-call-based IDS. Many evasion strategies for avoiding detection have been identified. [89] presents a systematic analysis of these evasion strategies and introduces the notion of mimicry attacks. Afterwards, a lot of work has been done to overcome the weaknesses identified. The major focus of these approaches is to improve the accuracy of the profile for normal process behavior used for anomaly detection. With an inaccurate profile, the IDS has to be more tolerant to behavior that deviates from that predicted by the profile. Otherwise, excessive false positive will result from the misprediction of normal, valid behavior. Unfortunately, this tolerance can be exploited to the

attacker's advantage. With a more accurate profile, the IDS can be stricter in its enforcement and mark any slight deviation from the normal profile as an intrusion.

Among the work done in this direction, [88] is one of the most exemplary. [88] is the first to propose the white-box-profiling technique; instead of treating the process being profiled as a black box, we can build the profile based on analysis of the corresponding program. They have proposed several techniques for white-box profiling, varying in the accuracy of the profile, as well as the efficiency of run time monitoring. If the analysis is done correctly, white-box profiling guarantees zero false positive. As a result, we can avoid the false-positive-false-negative tradeoff mentioned above. However, high profile accuracy comes at the cost of higher complexity in runtime monitoring. Some of the most accurate profiling techniques proposed in [88] make it extremely difficult for the attackers to evade detection. Unfortunately, the monitoring overhead based on these profiles is likely to be high, owing to the nondeterministic nature inherent in profiles generated by program analysis. In general, monitoring in this way has extremely high complexity, and is so slow that it is impractical for monitoring in real time. Requiring the availability of source code is another major drawback of this work. This makes it impossible to apply their techniques to commodity software.

Some work [23, 30, 31, 50, 96] has been done in overcoming these two drawbacks. To tackle the problem of high monitoring overhead, some tried to optimize the profile generated. There are also proposals for the monitoring of

other process characteristics that allows the differentiation of states that are seemingly the same. Some others attempt to instrument the corresponding program so that it will report the needed context information during execution time. The problem of unavailability of source code is to be solved by binary code analysis and binary code instrumentation.

Also, as is pointed out in [89], both input arguments and return values of system-calls are ignored in many system-call-based anomaly detection systems. Efforts to utilize the input and output of system-calls in anomaly detection are seen in [48, 47, 30, 31, 82], while [29] takes this idea further by considering information from the system environment (e.g. configuration files, command line input, environment variables). In addition to improving both profile accuracy and monitoring efficiency, many of these works propose new kinds of inputs for anomaly detection (e.g., return address, call stack information[23]).

System call arguments are also used in anomaly detection systems that do not follow the approach of [88]. In particular [48, 62] both build separate normal model for the input arguments associated with each individual system call. In [48], the model built is "context-free", i.e. all invocations of the same system service share a common model, while [62] improves the accuracy by constructing a different model for the different context (defined by the sequence of return addresses on the call stack) under which a system service is requested. A point that is worth nothing is that the IDSs in [48, 62] fits the definition of LAIDS. However, IPSs resulting from naively applying them

to the LAIDS/LIDS framework may not be very interesting (countermeasures may only identify system services invoked by the detected attack, and thus are largely payload specific). Nonetheless, the same trick we use to make random-inspection-based IDS useful under the LAIDS/LIDS framework may be applied to [62] (see Sect. 6.2), and the resulting IPS will generate countermeasures that identify the location and context of the last system call made before the control hijacking occurs.

While system-call based anomaly detection advance through using models generated by analyzing the source/binary code of the protected program, methods of developing mimicry attacks also grow more sophisticated. For example, [32] models how different system calls affect the state of the system and use model checking to discover mimicry attacks that achieve the attacker's goal. On the other hand, [46] shows methods for the attacker to regain control after invoking system services with crafted return addresses on the call stack that will evade IDSs like [23, 31]. Finally, [69] shows that patient attackers can achieve some non-trivial attack goals without actively issuing any system calls, instead, these attackers simply wait for the attacked process to invoke system services related to I/O and modify their arguments which are not monitored by any existing systems.

Due to the persisting vulnerability against mimicry attacks in system-call based anomaly detection, some have proposed new mechanisms for monitoring / observing the behavior of protected system. For example, [3] proposed to use binary analysis to identify all possible targets of every control transfer

in a protected program. Binary instrumentation is then applied to add code to check at run-time that each control transfer only jumps to allowed targets identified in the analysis phase. However, a point worth noting is that [3] assumes the analyzed and instrumented binary code is generated by the Vulcan compiler [21], which is designed specifically to allow binary instrumentation, and it is unclear whether the proposed analysis / instrumentation is feasible for legacy code or code generated by other compilers. In general, we believe a solution that requires compile-time information will not be very feasible for the Windows environment where applications from different vendors are deployed, since many software vendors will deliberately strip such compile-time information from the binary for commercial interest.

The work in [75] is similar to that in [3], but relies on dyninst [38] to perform binary instrumentation at run-time. Finally, [9] performs reaching definition analysis on the binary code, and employs binary instrumentation similar to that in [3] at each read instruction to make sure the value being read is "defined" by the right source. Once again, [9] assumes the analyzed and instrumented binary code is generated by a specific compiler (Phoenix [2]) that is designed to facilitate such analysis/instrumentation.

4.1.3 Specification Based

Similar to anomaly-detection, specification-based IDSs detect intrusion by identifying behavior that deviates from a normal model. However, in specification-based IDSs, the normal model is manually input by a human

expert. In fact, Program Shepherding, described in Sect. 3.3.1 is an example of specification-based IDS, in which rules about the targets of normal control transfers are manually specified. As we will see, our prototype for random-inspection-based IDS falls into the same category. Other examples of specification-based IDSs include [87, 73]. Finally, the popular DEP (data execution prevention) on Windows is an example of specification-based intrusion detection that is implemented in hardware; with the support from the CPU, the DEP prevents code execution from data pages, such as the default heap, various stacks, and memory pools. We note that our prototype random-inspection-based IDS can be seen as a probabilistically checking/enforcing the model employed in the DEP. Unfortunately, as we will see in Chapter 5, the protection provided by the DEP technology is far from complete. We find that it is not uncommon for benign applications to execute dynamically generated code which is located in data space. In order to avoid false positives, the current solution is basically to turn off DEP for these applications. In Chapter 5, we will demonstrate a remedy to this problem by extending our prototype random-inspection-based IDS. Finally, we stress that our prototype for random-inspection-based IDS only demonstrates one normal-model that can be checked/enforced by our approach, there are many other possible models that we can use on the monitoring mechanism provided by this approach.

4.2 Design Overview

Our original goals of proposing random-inspection-based IDSs were to solve two problems with the then popular system-call-based IDSs:

1. Their inherent vulnerability to mimicry attacks: we believe this vulnerability roots from the monitoring mechanism, i.e. the interposition of system calls, itself. In particular, we believe this monitoring mechanism is too predictable to the attackers, and makes it very easy for the attacker to manipulate what the IDS "sees". Furthermore, the monitoring of the system-call interface is also too passive and allows attackers to entirely avoid the IDS by not invoking any system services. Even though it is argued that the attackers MUST issue some system calls to achieve any non-trivial goal, we note that the passive nature of the IDS allows the attacker to learn about the attacked system / environment before starting the real attack activities.

2. Their being non-portable for the widely deployed Windows systems: the proprietary nature of Windows (e.g. the common use of dynamically generated code documented in Chapter 5, modification of return addresses on the stack within the called function, and obfuscated code generated to defeat binary analysis) can make it very difficult for binary analysis techniques to extract the kind of accurate model used by many advanced system-call-based IDS or systems like [3, 75, 9]. In fact, [3] explicitly assumes "non-writable code", and thus cannot handle ap-

plications that use dynamic code generation. A similar problem faces dyninst, the binary instrumentation tool used in [75]. The extensive use of DLLs on Windows system also means even if we successfully perform binary analysis on the various software modules, putting them together to form a complete model of a process may not be easy. Furthermore, the constant update on Windows environment will also create the need to constantly update the normal model obtained from a training phase or binary analysis, which presents another practical problem for using system-call-based IDSs on Windows. Finally, we note that the need for keeping a detailed model for each protected process in some advanced system-call-based IDSs can be a trouble in using them for full-system protection: the model for all processes in the system can take up a non-trivial amount of memory.

To solve these problems, we proposed a new monitoring mechanism called random inspection. In a random-inspection-based IDS, we periodically stop the monitored programs in a preemptive and unpredictable manner. Every time a program is stopped, we observe its current state and determine if it has been compromised[1]. The preemptive nature of random inspection means the attackers will have no way to avoid their activities from being observed; as long as the compromised process is in an "illegal" state, there is always a non-zero probability that our IDS will detect the attack. Furthermore, the

[1]Using the terminology from Sect. 3.2.1, we'll call points where the monitored process is stopped for such checking the "inspection points".

attackers also cannot predict when a process' state will be checked, and thus cannot "cover their track" at the right time to evade the IDS.

4.3 Core Random Inspection

We implemented random inspection by making use of a common hardware feature called performance counter. Performance counters are hardware registers that can be configured to count various processor-level events (e.g. cache miss, instruction retirement, etc). This facility is mainly designed for high-precision performance monitoring and tuning. Since events are counted by the CPU in parallel to normal operations, we can expect very low overhead for the counting. Furthermore, the CPU can be configured to generate an interrupt on any performance-counter overflow. As a result, by properly initializing the performance counters, we can stop the operation of the system after a certain number of occurrences of the event being counted. By resetting the counter to a random value after each counter overflow, we can make the timing of the next inspection unpredictable to the attacker. In our implementation, we generate this random value using a linear congruential generator, i.e. the random value is given by the formula: $X_{n+1}=(aX_n+\mathrm{c}) \bmod \mathrm{m}$, where

1. X_n is the previous random number generated, with X_0 being the seed, and we periodically reseed / restart the sequence with the value of a hardware timestamp (that counts the number of clock cycles since the last reset);

2. m is a constant that controls the frequency at which the interrupt occurs. In the following, we will call $m/2$ the inspection frequency;

3. $0 < a < m$ and $0 < c < m$ are constants;

Also note that by implementing random inspection using a hardware feature, the monitoring is performed on a system-wide basis. However, by intercepting context switches in Windows, we can perform "per-thread" monitoring, i.e. we can have different inspection frequency for different thread, and more importantly, we can save and restore the remaining counter value at context switches, so that the performance counter is counting the events occurring within the current thread. This proves to be very useful in our implementation of the CIP framework. One final design decision for implementing random inspection is what event the performance counter should be counting. We made this decision based on the following criteria:

1. we want an event that occurs at high frequency in both normal and injected code,

2. we want this event unavoidable in the injected code.

The first criterion allows us more freedom in the choice of inspection frequency. The second criterion makes random inspection more robust: the attackers cannot evade inspection by avoiding the counted event.

For our implementation, we choose to count the instruction retirement events [2] that occur in user space. We believe this event satisfies the above criteria very well. Furthermore, by counting events in user space only, we guarantee that inspection will only occur in user space. This allows easier utilization of information collected at inspection points.

We implement our prototype system on a machine with an Intel Xeon CPU. We note that performance counters that generate interrupt on overflow is very common in CPUs nowadays[3]. Thus our idea is not limited to Intel CPUs. Furthermore, we find the use of this facility is limited to profiling software only, so our implementation will not disrupt normal system operation.

4.4 A Prototype Random-Inspection-Based IDS: WindRain2

After discussing how random inspection is actually achieved, we now show our implementation of intrusion detection under the random-inspection mechanism. In the following, we present the details of our WindRain2 system[4].

The most important component of the WindRain2 system is a device driver that runs on the Windows platform (tested on both XP with service

[2]Instruction retirement marks the completion of the out of order execution of an instruction and the update of processor state with its results

[3]On Intel CPUs, this feature is available on any processor of the P6 family or later, on AMD CPUs, this feature is supported starting from the K7 family.

[4]We called our original prototype presented in [11] "WindRain"; since then, we've re-implemented our prototype to add support for per-thread inspection and use more random values for the performance counters, and we call this improved version WindRain2

pack 0 and service pack 2). We have also written an application that loads the driver and displays data received from the driver in a timely manner (most importantly, notification about intrusions). The driver is responsible for setting up the system to perform random inspection, i.e., configuring the performance-counter facility and registering an interrupt-service routine to handle performance-counter overflow. This interrupt service routine is the part that actually performs intrusion detection. Finally, the driver also captures various events in the Windows system for bookkeeping. In particular, we use techniques similar to [36] to intercept context switches, so we can save and restore the performance counter value accordingly. We also handle creation and destruction of threads and processes using Windows callback mechanisms, so we can start the per-thread inspection for each new thread, and clean up our bookkeeping data structures for processes / threads that are terminated. For the inspection of threads that are created before WindRain2, they are started when the threads resume execution at the first context switch after the driver of WindRain2 is loaded.

On performance counter overflow, an interrupt is generated and the interrupt service routine registered will be called. The interrupt service routine starts by resetting the performance counter with a random value. It will then clear some flags so that the counter can start upon return to the user space. After that, the real intrusion detection starts. Among the arguments passed to the interrupt service routine are the PC (program counter) value of the interrupted instruction as well as the values of the ebp and esp registers

("frame pointer" and "stack pointer") at the time of interrupt. Based on these values, WindRain2 will perform the following two checking:

1. determine whether that PC value corresponds to a memory location that holds code or one that holds data; if the PC value points to a data region, WindRain2 will report it as an intrusion.

2. based on the saved value of the ebp and esp registers, WindRain2 will walk the call stack and retrieve the last n return addresses (where n is configurable) saved on it; for each return addresses thus found, WindRain2 will check whether it points to data space, and if it does, WindRain2 will report an intrusion when the offending return address is used in a return instruction.

The implementation of the first checking is basically the same as described in Sect. 3.3.2. In the following, we will detail how we perform the second checking.

4.4.1 Checking Return Addresses

In most functions, the function prologue starts with the instructions "push ebp; mov ebp, esp", with the value of the register ebp kept constant until the function returns. As such, the ebp register basically points to a linked list of activation records, i.e. the memory location pointed to by the ebp register stores the saved ebp value of the caller to the currently executed function, and the location pointed to by this saved value points to the ebp

value of the caller's caller. Furthermore, the address that immediately follows the saved ebp value is the return address that points to some call site in the function that "owns" the saved ebp value. As such, we can "walk through" the call stack by going through the linked list of saved ebp values, and retrieve all the return addresses that have been saved on the call stack. We can then check each of these addresses to determine whether they point to code space or data space using the same method as we check the PC value.

However, there is one problem with our stack walking mechanism: not every function in Windows is "normal" and starts with the normal function prologue (this problem is documented in [39]). In other word, there are functions that do not save the ebp value, or that alter the ebp value during their execution. If one of these functions (say function X) calls another normal function Y, which eventually results in the current function being called, the linked list of ebp is broken. In other word, we'll walk through the call stack and retrieve the ebp value saved when the abnormal function X calls function Y; if we follow this retrieved ebp value, we'll get a pointer that does not point to any valid stack frame, and read off a wrong return address from the stack (we can detect the error if the address is outside the stack). As a result, such abnormal functions can result in false positives in WindRain2 if we take the output of our stack walk as the actual list of return addresses saved on the stack.

We tackle this problem by performing an extra checking on the first offending return address we found on the stack. In particular, we want to

make sure it is really going to be used in a return instruction. We confirm this by configuring the CPU's breakpoint registers so that an interrupt will be generated when the "instruction" at the offending address is executed. Thus, if the offending address is actually a return address saved on the stack, our breakpoint will be set off when this address is used in the corresponding return instruction. On the other hand, if the offending address does not point to a valid instruction, the breakpoint will never be set off. Since the CPU's hardware registers is a limited resource, we implement an expiration mechanism for the breakpoints thus set. Under this mechanism, we will remove the breakpoint at the 5-th inspection point after it has been set.

4.5 Analyzing the Detection Rate of Random-Inspection-Base IDSs

Before we present the results of our experimental evaluation on WindRain2, we first analyze the probability of WindRain2 detecting different code injection attacks. We note that similar analysis can be applied to predict the performance of other random-inspection-based IDSs.

The simplest way to perform this analysis is to consider inspection as a Poisson process, and calculate the probability that one or more inspection will occur while the compromised process is in an illegal state, where an illegal state is any state that will result in the IDS generating an alert should an inspection occur in that state. For WindRain2, an illegal state is one where the CPU is executing the injected code in data space, or executing functions

called on behalf of the attacker so that the return address to the injected code can be retrieved by the stack walk.

Using the definition in Sect. 4.3, in WindRain2, we perform an inspection once every $m/2$ user space instructions executed (on average). Assume the compromised process executes y instructions in some illegal states, the probability of detection is then $P_d = 1 - P(0) = 1 - e^{-\frac{2y}{m}}$.

The above analysis does not assume continuous execution in illegal state. Therefore the probability computed is valid even if the injected code calls some function that breaks the chain of saved ebp values, or the return address to the injected code is too deep in the stack to be reached by the stack walk. Another very important point is that the above analysis is only valid if the attacker cannot predict when the next inspection will occur. Otherwise, it is (in theory) possible for the attacker to evade detection by calling certain library functions when an inspection is expected, so the return address saved on the stack is not reachable by the stack walk.

4.6 Evaluating WindRain2: Detection Rate

In this section, we will present our evaluation on how well WindRain2 detects attacks. Instead of testing our prototype against real attacks found in the wild, we've performed our experiments using two shellcode provided by the metasploit framework [1], namely the "windows /exec" and "windows/shell_bind_tcp" shellcode. For the first shellcode, we've configured it to execute "notepad.exe", then exit the compromised process. For the sec-

ond shellcode, once it spawned a command shell that receives input from the network, we send in the "exit" command to close the shell. To launch the "attacks" in our experiments, we have also written a small program that will copy the shellcode into the heap and start executing it.

Since the detection rate of WindRain2 is entirely determined by the number of instructions executed in an illegal state, we believe our first shellcode will present WindRain2 with a worst case scenario; the shellcode is significantly simpler than any that can be found in common worms (e.g. MSBlast, Sasser), and we believe it is so simple that it won't be of much use in real attacks. As for the second shellcode, it is very similar to that used in MSBlast and Sasser (both in functionality and in implementation).

We have measured the detection rate of WindRain2 against the two shellcode above, while it is running at various average inspection frequencies. We have also experimented with retrieving different number of return addresses from the call stack at each inspection point. To illustrate the value of checking return addresses on the call stack, we have also studied the detection rate when WindRain2 does not walk the stack but checks only the address of the current instruction. Finally, for each tested configuration, we repeat the "attack" 100 times and count how many times WindRain2 detect the attacks. Our results are presented in Fig. 4.1 and 4.2.

From the results in Fig. 4.1 and 4.2, we see that WindRain2 can reliably detect attacks that employ the first shellcode with an average inspection frequency of once every 50,000 instructions or higher. As for the second, longer,

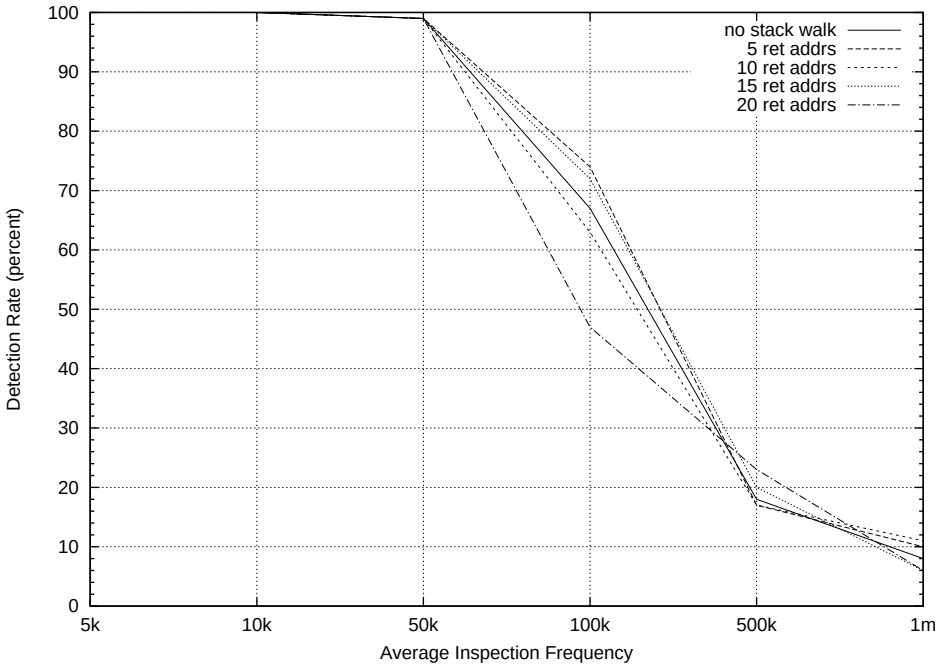

Figure 4.1: WindRain2's capability to detect an attack that create a process and exit the compromised process when operating at various inspection frequency and retrieving different number of return addresses from the stack at each inspection. Inspection frequency is measured by the average number of instructions executed between two inspections.

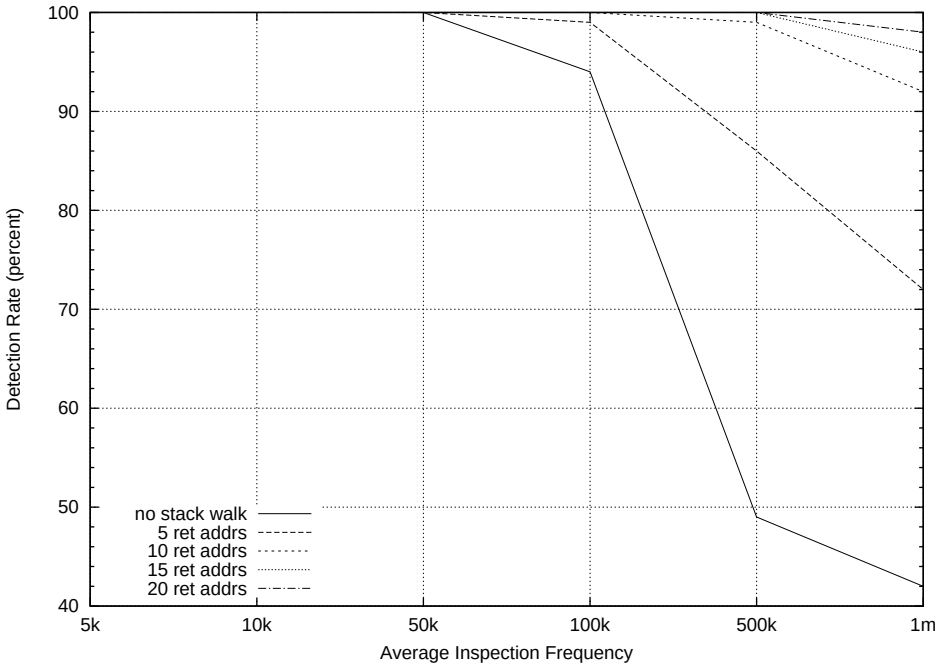

Figure 4.2: WindRain2's capability to detect an attack that spawn a command shell and bind it to a port, while operating at various inspection frequency and retrieving different number of return addresses from the stack at each inspection. Inspection frequency is measured by the average number of instructions executed between two inspections.

more realistic shellcode, WindRain2 can maintain a 100% detection rate while performing random inspection once every 100K to 500K instructions on average. We also note that checking return addresses appears to be of little value against the first shellcode, but significantly improves detection rate against the second shellcode. This is because the first shellcode only calls two library functions (and thus stack walk only improves detection rate if inspection occurs while these two functions are executed on behalf of the attack), while the second shellcode requires 10 calls to library functions. The benefit of performing deeper stack walk is more obvious when WindRain2 is performing low frequency inspection. This is because at a high inspection frequency, multiple random inspections may occur during the execution of the shellcode; thus, a high detection rate can be maintained even if some of these inspections miss the attack due to insufficient stack walk. On the other hand, when WindRain2 is performing low frequency inspection, any given inspection may be the only one that occurs during the course of an attack; as a result, a deeper stack walk at each inspection point can make a significant impact on the detection rate. Finally, we note that the detection rate of WindRain2 appears to be significantly higher than that of WindRain (as presented in [11], an average inspection frequency of once every 3600 instructions is needed to maintain a 100% detection against Sasser, which uses a shellcode similar to the second one we've tested). We believe such improvement comes from the use of per-thread inspection, as well as more random values for setting the performance counter (which determine when the next inspection occurs).

4.7 Evaluating WindRain2: Performance Overhead

In this section, we present the results of our experiments that study how much performance overhead is incurred on normal processes when WindRain2 is performing random inspection at various frequencies and retrieving different number of return addresses from the stack. All the experiments reported in this section, as well as in the performance evaluations in subsequent chapters are performed on a PC with a Xeon CPU and 1GB RAM.

For our experiments, we've used the following 4 benchmarks:

1. gzip: a CPU intensive application that belongs to the SPEC benchmark suite, we used a 8MB file for input in our experiments.

2. javascript benchmark: a benchmark for a browser's Javascript engine which involves operations like mathematical calculations, DHTML, string manipulation, image swapping, table manipulation, page content manipulation and window management; this is one of the tests used in [41] to compare the speed of different browsers. Since we were not aware of the newer version of the benchmark until we've finished most of our experiments, the results presented below are based on the 2006 version of the benchmark.

3. CSS benchmark: another benchmark listed in [41] for browser speed comparison, which measure the CSS rendering speed of a tested browser.

4. Apache "flood": a web server benchmark which generate HTTP requests

based on an input profile and measure the time it takes for the server to service the requests, we used it on our installation of the Apache web server, with a version of the example profile "round-robin" that is modified to retrieve three of the Apache "welcome" pages of size 2KB, 4KB and 7KB respectively.

In addition to studying the performance overhead incurred at various configurations of WindRain2 tested in the previous section, we have also studied the performance overhead incurred by an "empty" version of WindRain2 that does not perform any analysis, but simply handle the interrupts caused by performance counter overflows, and set up the environment for performing the real analysis (mostly to allow page fault during the analysis).

Our preliminary experiments with the Apache "flood" tool show that the performance overhead reported by this benchmark is very low even when WindRain2 is performing random inspection at a very high frequency and is configured to retrieve the maximum number of return addresses at each inspection; the slowdown observed in the result of Apache flood is also order of magnitude lower than that in the other benchmarks. In fact, the overhead recorded by Apache flood is so low that we cannot observe any trend in how the performance overhead changes when we increase the inspection frequency and the depth of the stack walk performed by WindRain2; this is because any change in the overhead is insignificant when compared to the fluctuation in the time it takes to perform the benchmarked operations. We believe the reason why the performance reported by Apache flood is unaffected by WindRain2 is

that the benchmarked operations are too I/O intensive, while WindRain2 slows down a system by making the CPU more busy performing the computation at each inspection point. Since I/O operations are still order of magnitude slower than CPU operations, the slowdown caused by WindRain2 will be completely overshadowed by the time it takes to perform the I/O activities involved. As such, we believe Apache flood is not a meaningful benchmark for measuring the performance overhead incurred by WindRain2 and do not present the results of our experiments on this benchmark. The results of the remaining of our experiments are presented in Fig. 4.3, 4.4, and 4.5

From the results presented in Fig. 4.3, 4.4, and 4.5, the following observations can be made

1. The most determining factor in the performance overhead incurred by WindRain2 is the inspection frequency, and the performance overhead incurred drops very quickly as we decrease the inspection rate. The explanation of such trend is simple: even for the most costly analysis performed at each inspection, if we perform it rarely enough, the overhead of the analysis will be minimal. This also explains why the extra cost of performing deeper stack walk is obvious at high inspection frequencies, but almost disappears when the inspection rate goes below once every 50,000 instructions.

2. By comparing the performance overhead of the "empty" WindRain2 and when WindRain2 performs actual intrusion detection, we see that

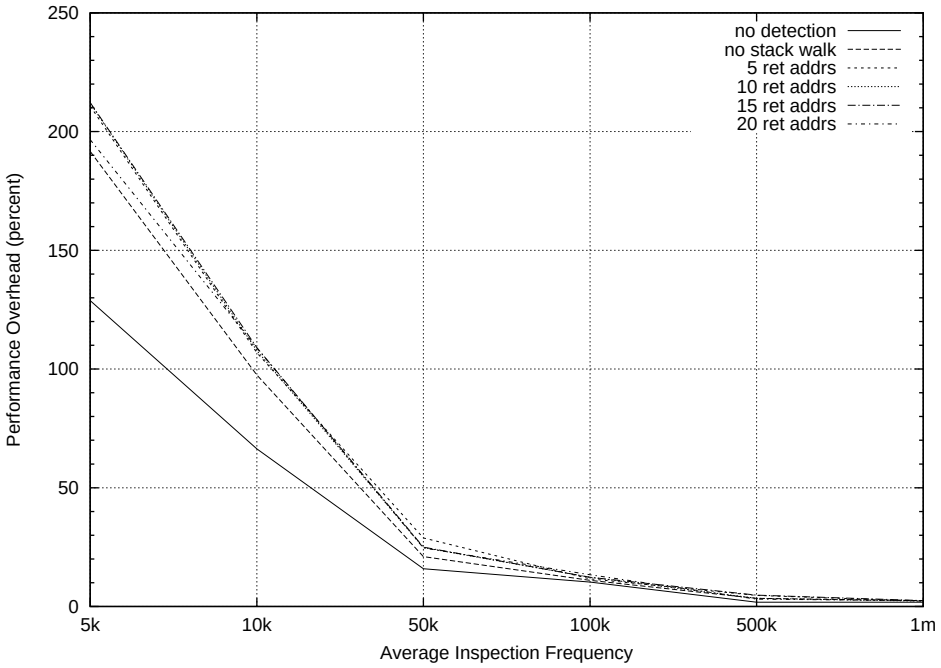

Figure 4.3: The performance overhead incurred on gzip when WindRain2 is performing random inspection at various frequencies and retrieving different number of return addresses at each inspection.

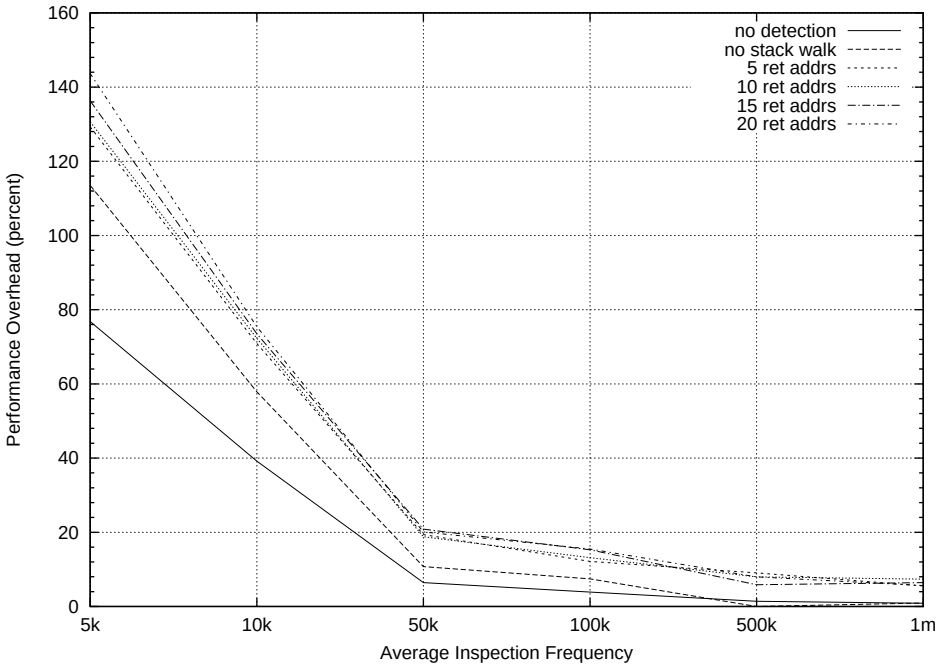

Figure 4.4: The performance overhead incurred on the Javascript benchmark when WindRain2 is performing random inspection at various frequencies and retrieving different number of return addresses at each inspection.

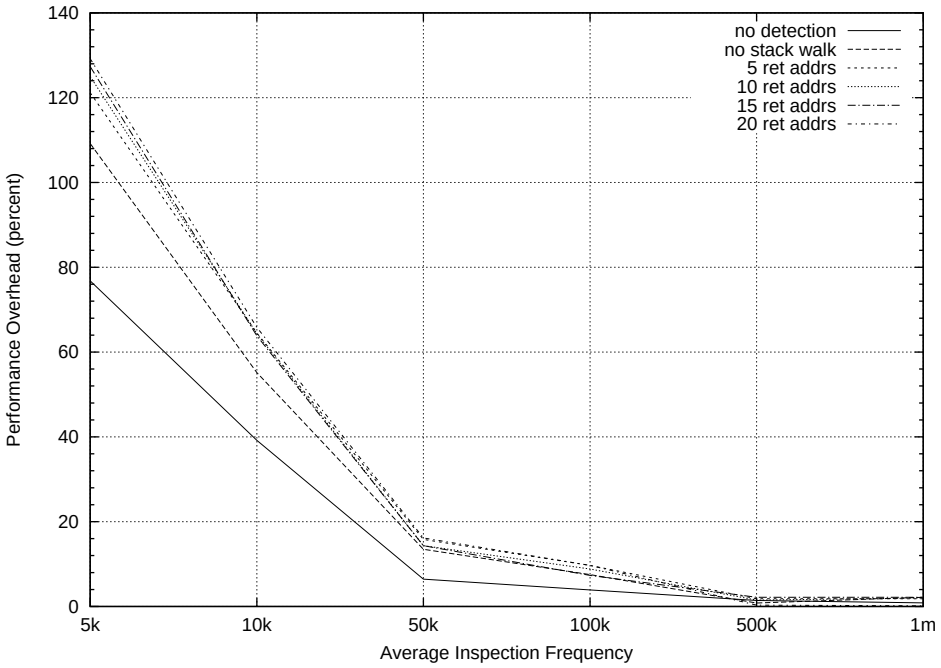

Figure 4.5: The performance overhead incurred on the Css benchmark when WindRain2 is performing random inspection at various frequencies and retrieving different number of return addresses at each inspection.

the cost of handling interrupts and setting up the environment for the analysis accounts for a very significant portion of the overhead. On the other hand, the cost of performing more sophisticated analysis is much less significant.

3. By comparing the results of WindRain2 and that of WindRain in [11], we find that WindRain2 incurs a significantly higher performance overhead when operating at high inspection frequency. We believe this is mainly caused by the difference in the hardware used for the experiments. In particular, we note that the CPU we used for the experiments on WindRain2 is amongst the least efficient in interrupt handling in all Intel CPUs (according to [59]), thanks to the long pipeline of the P4/Atom architecture. Thus, we believe WindRain2 will have better performance on newer CPUs (like Intel Core) which have shorter pipelines than the one used in our experiments.

4. When we consider the results of the experiments presented in both the previous section and this section, we find that WindRain2 can reliably detect attacks that create a single process on the victim system with a less than 25% overhead (performing one inspection every 50,000 user space instructions executed). As for attacks that spawn a shell and listen for commands over the network, WindRain2 can achieve a 100% detection even when inspections occur, on average, once every 500,000 instructions, and incurs a performance overhead of less than 10%. Our

analysis also shows that it is very cost-efficient to perform deep stack walk when WindRain2 is operating at low inspection frequencies; the improvement in detection rate achieved through deeper stack walk is significant, while its cost is minimal.

4.8 Conclusions for Random-Inspection-Based IDS

In this chapter, we presented the idea of random-inspection-based IDS, as well as our prototype, WindRain2, that implements the idea. Our evaluation of WindRain2 makes us believe that it can detect any realistic injected-code attack that exists in the wild, while incurring a less than 10% overhead on the protected system. Furthermore, with WindRain2 checking both the address of the currently executed instruction and return addresses retrieved from the call stack, we believe WindRain2 will have a significantly larger window for detecting attacks when compared with its predecessor, WindRain, which checks only the current program counter value.

The stack walk in WindRain2 also makes it very difficult for the attackers to "jump out of" an illegal state while achieving their attack goals. Even if the attackers manage to locate the library functions they need, calling these functions (and jumping into code space) will not necessarily prevent detection; if the functions called save the ebp register in their prologue, the execution will remain in an illegal state until the call stack grows too deep for WindRain2 to reach the return address that points to the injected shellcode. Furthermore, once a library function is called, the state of the compromised thread seen by

WindRain2 is beyond the control of the attacker, and is entirely determined by the callee function. In other word, if the attacker needs a library function that keeps a proper stack frame and executes many instructions while maintaining a very shallow stack, he/she will have no choice but remain in a state susceptible to detection over a long period of time. Such difficulty in getting the compromised thread into a state acceptable to WindRain2, coupled with the non-determinism in the occurrence of random inspection, will make it very hard to evade detection.

Admittedly, it is not impossible for attackers to develop techniques that allow them to call library functions while having the return address to the injected code concealed most (or all) of the time. In fact, a technique of such flavor has been presented in [46]. However, we note that [46] only focuses on concealing the illegal return address at the point where system services is invoked; afterwards, the return address will be restored to its proper value that points to the injected code. Thus, it is unclear how effective the technique in [46] is in reducing the detection window for WindRain2. Furthermore, in the context of collaborative intrusion prevention, such techniques for reducing the detection window of WindRain2 will only mean we need more hosts in the collaboration, and cannot render our defense entirely useless. As we will see in Chapter 6, it is quite unlikely for such technique to increase the number of hosts needed in the collaboration for a reliable defense to an unrealistic level; our experiments show that even for an extremely simple attack that execute 2 library calls, the number of hosts needed in the collaboration to defend against

it will be less than 1000.

One shortcoming of WindRain2 is that it cannot detect any existing code attacks, since they do not execute any code in data space. Though we believe such attack is still largely theoretical, when / if they become a problem in practice, we are confident that WindRain2 can be extended to perform more analysis on the return addresses retrieved from the call stack and reliably detect these attacks. For example, we can check not only whether a retrieved return address points to the code space, but also confirm that it points to an instruction following a call site[5].

Finally, we note that an IDS with similar design as WindRain2 may have a significantly lower detection rate when running on *nix systems. This is because, as noted in [81], system services provided by Windows are usually very limited as compared to *nix systems. As such, to perform the same operation, a process on Windows will need to execute many instructions in user space (usually in library functions), while a *nix system will perform most of the work in kernel space. This difference means that the stack walking performed by WindRain2 will be more useful in detecting attacks on a Windows system. Nonetheless, we note that the difference between Windows and *nix systems only lies in where the operations performed on behalf of the attacker are carried out; in order to achieve an attack goal, we believe a similar amount

[5]This checking was proposed in [43], and has been employed in our Lazy Shepherding system presented in Sect. 3.3; this extra checking will allow WindRain2 to detect most forms of existing code attacks that have been proposed (e.g. [74])

of processing is needed on both types of platforms. Thus, as a future work, we will consider the possibility of performing random inspection while the monitored process is operating in kernel space (the inspection will examine the user space state that the process was in before entering kernel). With such extension, WindRain2 will be suitable for both Windows and *nix systems.

Chapter 5

Data Execution in Benign Programs

In this chapter, we will evaluate the false positive rate of our prototype random inspection-based IDS, WindRain2. Since our prototype checks for execution of instructions in data space, the work presented in this chapter can also be seen as an evaluation of the protection provided by the DEP technology. We will also present an enhancement to our prototype IDS so that it can distinguish benign data execution from code injection attacks. With this enhancement, we believe WindRain2 can be used to protect many applications (e.g. Microsoft Word, Visual Studio, any application that uses Java/.NET) that cannot be covered by DEP, and detect attacks that will have to be ignored by DEP. We will start our discussion by presenting our experiments that study whether/how benign applications execute data and create false positives in our IDS.

5.1 A Study of Data Execution in Normal Applications

From our experiments in [11] as well as those in the previous chapter, we find that both of our prototypes for random-inspection-based IDS (WindRain and WindRain2) have no false positives for Windows system processes (pro-

cesses created during Windows startup). For this reason, we choose to focus our attention to user applications on the Windows platform. We also find that many Windows applications execute data in the form of ATL thunks [67]. However, as we've shown in [11], these thunks are simply two instruction snippets of code that are very easily recognized. In other word, we can eliminate these false positive cases by having the IDS recognize ATL thunks and ignore alerts generated by them[1]. Thus, in the following discussion, we ignore any data execution caused by ATL thunks.

For our experiments, we have studied 10 popular Windows applications (some of the most downloaded software according to "download.com", as well as some programs we uses). Since our tool for studying the behavior of these applications comes as a device driver and intercepts various events in the Windows system, it has obvious conflicts with various security products like antivirus or spyware detectors. As such, we only focus on non-security software. The list of software tested in our experiments is given in Table 5.1.

For each of the studied application, we monitor every control transfers made during the application's start up and shut down phase (i.e. we start the application, and then close it). To perform this monitoring, we've implemented a tool based on the "last branch, interrupt and exception recording" on Intel CPUs, which records the source and destination of every control transfers executed, and saves the addresses in a designated buffer in memory called the

[1]DEP on Windows XP and Vista handles ATL thunks in a similar manner.

Name	Rank	Description
Apache	N/A	Popular web server
Visual Studio 2005	N/A	Develop environment
Microsoft Word 2003	N/A	Word processor
Java	N/A	Runtime environment for the Java language
Orbit Downloader	8	Download manager
CamFrog Video Chat	9	Video chat software
You Tube Downloader	11	Download video from YouTube and convert them to other media formats
PrimoPDF	31	Print PDF to Windows applications
Download Accelerator Plus	46	Download manager
Paint.NET	3-rd among Image editing software	Image editing software

Table 5.1: Ten applications we have tested for our experiments. For applications that we found in download.com, we also listed their popularity ranking (as of the week ending on 18-th Jul, 2009). Note that many of the tested applications have higher ranking when we started our work in this chapter in January 2009.

branch trace store (BTS) buffer. An interrupt will be generated when the BTS buffer is full, and we will analyze each record in the buffer to see if the target of any control transfer is in data space. Since the above recording mechanism is a hardware feature that Windows is not aware of, it is by default a system-wide recording. To avoid the mixing of records from different processes, we need to intercept context switches and save/restore all the branch records.

The main reason we focus only on the start up and the shut down phase of the studied applications is because of the significant slow down caused by our tool, which makes it very difficult to further test any application manually (according to the Intel manual [37], the branch recording alone causes 20 to 40 times slow down, and the analysis performed by our tool can bring the slowdown up to 100 times). Even though our study appears extremely limited, for applications that are found to execute data in our experiments, the information obtained from the startup phase can be generalized to predict many of their normal behavior, and allow us to filter out all false positives in WindRain2 that are caused by most of these applications. Furthermore, we note that it is very possible to significantly optimize our tool, and employ some test generation / user input replay tools to automate the testing process.

In addition to finding out whether an application executes code that is located in data space, we are also interested in knowing where such code is located, what they look like, and what DLLs are responsible for generating such code. For this purpose, our tracing tool also intercepts heap creation and virtual memory allocation events. For each such event, we will record

the context (return addresses on the stack) of the heap creation / memory allocation.

Such context information is essential to our study of which DLL is responsible for generating the data being executed. In particular, once we find an application executing code in data space, we can retrieve the context under which the memory region involved is created (in all the tested applications, writable memory regions that contain code are always created through the intercepted functions). With the retrieved information, we repeat our experiments, but pay special attention to all memory regions created under the identified contexts. In particular, we modify the corresponding page table entries (PTE) to mark these regions as read-only, and intercept all page faults generated by attempts to write these pages (by hooking the page fault interrupt). In the interrupt handler, we record the identity of the DLL responsible for the write operation, and restore the PTE of the page involved to make it writable. Before resuming normal execution, we also set up the performance counter to generate an interrupt at the next control transfer in the user space (using the same mechanism as in Sect. 4.3). This interrupt will allow us to "relock" the page that we just made writable. With the above monitoring mechanism, we can identify the DLL responsible for writing (almost) any given byte in the pages that are of interest. Finally, when we find a jump to data space while processing records in the BTS buffer, we can identify the DLL that generates the code at the target of the control transfer.

To understand what kind of code are being executed in data space,

our tracing tool can also dump all the instructions involved in data execution. In particular, starting with the branch record for the jump into data space, up until the record for the jump back to code space, we copy all the bytes between the target address of one record and the source address of the next. As such, we will have copied the code in all the basic blocks for the code that is executed in data space. In the following, we'll summarize the findings of our experiments.

We start by reporting that 6 out of 10 tested applications (namely, Microsoft Word, Visual Studio, Paint.NET, PrimoPDF, DAP and Java) execute code in data space.

Our analysis also shows that in some cases (e.g. Microsoft Word), the code being executed in data space appear to be merely wrapper functions that push parameters onto the stack, call a function, cleanup the stack and return. Nonetheless, we believe it is infeasible to handle false positives caused by such wrapper functions using the same method used for ATL thunks. First of all, we did observe some variety between the different wrapper functions captured by our tracing tool; as such, building signatures to identify all such functions may not be easy. Furthermore, we believe it is entirely feasible for attackers to build non-trivial shellcode that are disguised as a series of such wrapper functions. In fact, as shown in [74], the capability to set up function calls is all the attackers need to perform arbitrary computation in their shellcode. As a conclusion, we believe attempts to filter out data execution caused by what we believe to be "wrapper functions" using some signature-based mechanism will

have non-zero false positives and can be exploited to the attackers' advantage. A similar approach will be even more futile in handling applications that use just-in-time compilation and execute code far more complicated than wrapper functions in data space.

We also observe that for all 6 applications that are found to execute code in data space, only a very limited number (two at most) of DLLs are responsible for the generation of the offending code. In particular, in PrimoPDF, Paint.NET and Visual Studio, the offending code are generated by mscorwks.dll and msvcr80.dll (we believe these two libraries to be the portion of the .NET framework responsible for just-in-time compilation). For Java, the offending code are generated by jvm.dll. For Microsoft Word the culprit is MSO.dll, while in DAP, code being exeucted in data space are not generated by a DLL, but by the executable program DAP.exe. We also note that the writable memory regions that contain code in PrimoPDF, Paint.NET and Visual Studio are all allocated under the same context, which further suggest that the execution of data in these applications are caused by the JIT under the .NET framework.

5.2 Handling False Positives in WindRain2

Our findings in the previous section implies that our prototype random-inspection-based IDS, WindRain2 will have many false positives when it is used to protect some of the applications we have tested. Obviously, DEP faces the same problem as WindRain2; before we present our enhancement

to WindRain2 to handle these false positives, let's see how DEP on Windows solves this problem.

DEP on Windows basically avoids false positives by turning off the protection for many programs. For example, according to [57], in Windows Vista, "DEP automatically monitors essential Windows programs and services. You can increase your protection by having DEP monitor all programs." In other word, DEP only covers "essential Windows programs and services" and is not by default turned on for all programs on Windows Vista. We also noticed that some of the applications tested in Sect. 5.1 try to work with DEP by declaring heaps that hold dynamically generated code as executable. In fact, except for DAP, all the 6 applications that are found to execute data in Sect. 5.1 declare the memory regions for holding the offending code as "executable/readable/writable". However, we believe this is defeating the purpose of having DEP; in particular, attackers can write their shellcode to these executable heaps and carry out an injected code attack without being detected by the DEP. Though we are not aware of any attack of this kind, it is at least theoretically possible through exploiting a heap buffer overflow or a format string vulnerability that allows the attacker to "write anything anywhere" (e.g [27]). If the vulnerability can be reliably exploited multiple times (through attacking different threads in a process), an attacker may be able to build a small shellcode in the executable heap by exploiting the vulnerability repeatedly, writing a small part of the shellcode each time. The shellcode thus built will in turn copy the remaining of the "real payload" into these executable heap

and execute them. A similar idea has been studied in [14].

We believe the solution to the above problems lies in a finer-grain policy concerning data execution. From our experiments in Sect. 5.1, we see that not only are dynamically generated code usually stored in dedicated heaps, they are also generated by dedicated libraries. Thus, we propose to distinguish benign data execution from injected code attack based on the library that writes that code. In other word, our policy concerning data execution will not only specify which memory region is allowed to hold executable code, but also identify all the DLLs that are allowed to put executable code in these memory regions. We believe this is a very intuitive approach, since it is quite unnatural to have the capability of generating executable code distributed over a large number of DLLs. This is especially true if dynamically generated code is used in the context of JIT; in this case, we will expect the code responsible for JIT to be located in a few DLLs that are highly optimized for this task. Even if data execution is used as a means of obfuscation, it is possible that existing obfuscation techniques will be compatible with our conjecture. Furthermore, once/if our proposed solution is widely adopted, we believe future obfuscation techniques can be designed to restrict code generation within a small number of DLLs. The knowledge of which DLL is responsible for dynamic code generation is not very useful in reverse-engineering, since a DLL can contain many different functions, and not all of these functions are responsible for code generation.

The proposed solution also allows a lot of flexibility in the specification

of policies. While we can have restrictive policies that enumerate exactly the set of DLLs that are responsible for dynamic code generation, we can also have more relaxed policies that allow all DLLs in a certain directory to generate code. The policies can be further relaxed to identify all the DLLs that must not be used for code generation (e.g. ntdll.dll, kernel32.dll).

Finally, we argue that the proposed solution should have very good resilience to the attackers' attempts of evasion; while existing exploitation techniques may allow much freedom in what data structures to overwrite and what value to overwrite them with, the instruction that actually performs the memory contamination is fixed for the underlying vulnerability. For example, in a heap buffer overflow, the "offending" instruction will most likely be in one of the heap management functions in either ntdll.dll or kernel32.dll; on the other hand, for a format string vulnerability, the "offending" instruction will be in the vulnerable function that does not handle format strings properly. As such, if the "offending" instruction involved in a particular vulnerability is not in a DLL that's allowed to write executable code, the attacker may have a hard time evading detection.

5.3 Implementing the False Positive Filter

Our implementation of the above idea to distinguish benign data execution from injected code attack is very similar to the part of our tracer program that identifies which DLLs are responsible for writing code that reside in data space. In particular, our policies concerning data execution is built on top

of the information regarding the context of creation for the different writable memory regions that contain executable code. For each such context information collected by our tracer program, we associate with a list of DLLs that are allowed to write executable code to memory regions created under the specified context. In other word, our policies identify writable memory regions that may contain executable code by the context at which they are created, and specify all the DLLs that are allowed to write executable code to these memory regions. If we find code that are not created by the allowed DLLs being executed, we generate an alert.

To enforce the above policies, we intercept all heap creation and virtual memory allocation events. Once we find a heap being created or a range of virtual memory being allocated in a context targeted by our policies, we mark the memory regions concerned as read-only. Attempts to write to these memory will result in a page-fault. As we did in our tracer program, we intercept the handling of such page faults; however, our processing in the interrupt handler is slightly different. Instead of recording the identity of the DLL responsible for the write attempt, we determine whether this DLL is allowed to generate code in the corresponding memory region. If the write operation is performed by a DLL that is not allowed to write executable code to the memory region involved, we record both the addresses targeted by the write, as well as the values being written to those addresses. On the other hand, if the DLL performing the write operation is allowed to generate executable code, we will clear any write records concerning the addresses targeted by the

write. In both cases, we change the protection for the memory region involved to allow the write to proceed, and setup the performance counter to generate an interrupt so we can mark the "unlocked" page as read-only later.

When data execution is detected, we try to retrieve the write records for the addresses that contain the code being executed. If such write records exist and the values in the record are the same as the current content of the addresses involved, we generate an alert. Note that we did not immediately report an offense when a DLL that is not allowed by our policies write to a monitored memory region, but wait till we can confirm that the values written by these offending operations are actually executed as code. This allows us to handle applications that use the targeted memory regions to hold both code and data, or those that manage these memory regions as heaps. In fact, we find that such provision is necessary to work with programs under the .NET framework. Our experiments in Sect. 5.1 show that memory regions holding dynamically generated code for such applications are constantly written by code in ntdll.dll; however, we also found that values written by ntdll.dll are never executed as code. Also note that we record values written by offending write operations so that we won't generate false positives even if we miss some writes from DLLs that are allowed to generate executable code at addresses which are previously written by other DLLs. In this case, (hopefully) the recorded values at the addresses involved will be different from their current values (which correspond to the code written by allowed DLLs).

5.4 Evaluating the False Positive Filter

In this section, we'll evaluate the false positive filter proposed above. We have implemented our filter as an enhancement to WindRain2, and we will determine:

1. does our filter stop WindRain2 from generating false positives for applications that execute code in data space during their normal operations?

2. can WindRain2 detect attacks against such applications that write shellcode to memory regions containing dynamically generated code and execute the shellcode from there?

3. how much performance overhead is incurred by tracing writes to pages identified by our policies as containing dynamically generated code?

5.4.1 Reducing False Positives in WindRain2

We evaluate the false positive rate of WindRain2 after the proposed enhancement by testing it against the 6 applications that are found to execute data in Sect. 5.1. As opposed to only testing their start up and shut down processes, we tried to execute various normal operations of the tested applications while WindRain2 is performing random inspection with an average frequency of once every 5000 user instructions executed, retrieving 20 return addresses from the stack at each inspection (the configuration we've tested in Sect. 4.6 that results in the highest detection rate, and thus expected to have

the highest false positive rate). The operations we have performed during the experiment with each of the 6 applications are listed as follow:

1. Visual Studio: creating a project, typing in code, modifying properties for the project (adding "include directories" and "dependencies"), building the project, building the apache apr, apr-util and apr-iconv packages.

2. Microsoft Word: writing up a journal paper review, formatting the review, inserting tables, pictures, symbols, date and time into a document, spell checking and grammar checking.

3. Java: running the SPECjvm2008 benchmark suite.

4. PrimoPDF: converting a Word document into PDF.

5. Download Accelerator Plus (DAP): downloading a single HTML file from the Apache website[2], downloading 10 ".exe" files from the above URL, using the "download all" function of DAP, used the FTP function of DAP.

6. Paint.NET: drawing, resizing the canvas, undoing actions, rotating the drawing applying various effects (e.g. oil painting, outlining) to a large image.

We report that WindRain2 generates no alerts during all our experiments listed above. We also note that except for DAP, the information (i.e.

[2]URL: http://archive.apache.org/dist/httpd/binaries/win32/

context at which code-containing memory regions are created and the identity of the DLLs responsible for putting executable code in those regions) collected by our tracer program during the start up and shut down phase of the tested programs are sufficient for generating the policies used in our experiments. As for DAP, when we try to use the "download all" function, we detect data execution in memory regions created in a context never seen before. Furthermore, we find that the DLL responsible for writing the code in these memory regions is not a properly loaded library, but appears at changing addresses. Nonetheless, this DLL always resides at a fixed offset from the beginning of the memory region involved, and we modify our policies for DAP to identify it based on the above observation. We believe this is the result of obfuscation technique used in DAP. After this addition to our policy, we can perform all tested operations in DAP without causing any false positives.

5.4.2 Detecting Execution of Data Written by the Wrong DLLs

We have also tested WindRain2's capability to detect a hypothetical attack that exploits a "write anything anywhere" vulnerability to build a shellcode in a memory region that holds dynamically generated code and later hijack the control to execute the shellcode thus built. For our experiments, we use Paint.NET as our target application, and intercept the "rtlfreeheap" function (using techniques similar to [36]) so that a small stub placed at address 0x7ffe0470 will be executed every time rtlfreeheap is called in Paint.NET (since the address is not writable, WindRain2 will not consider the execution

of the stub as an attack). At each invocation, the small stub will either try to write 4 bytes of a tested shellcode to one of the code-carrying memory regions of Paint.NET, or transfer control to the beginning of this shellcode on the last invocation after all bytes of the shellcode have been written. We note that this is quite similar to an attack exploiting a heap overflow vulnerability that allows the attacker to "write anything anywhere", except that the write operation in real attacks will most likely be carried out by instructions in the heap management code. To better simulate a real attack, we programmed the stub above to perform real attack operations (i.e. write shellcode/execute shellcode) on every 20-th invocation, since we believe each exploitation of a heap overflow will allow 4 bytes to be written (one invocation to our stub), and various exploitations will be separated by at least 20 executions of the instruction responsible for contaminating the memory. For our experiments, we've tested the enhanced version of WindRain2 against attacks that use two shellcode tested in Sect. 4.6, with WindRain2 performing random inspection at various frequencies, and configured to retrieve 20 return addresses from the stack at each inspection. The results of our experiments are presented in Fig. 5.1.

For each experiment, we have also recorded the detection rates for illegal writes executed by the stub, and we find that WindRain2 detects all these write operations. This 100% detection rate of the illegal writes explains the similarity between the results in Fig. 5.1 and that in Sect. 4.6; WindRain2 has records that indicate all bytes in the shellcode are written by a DLL that

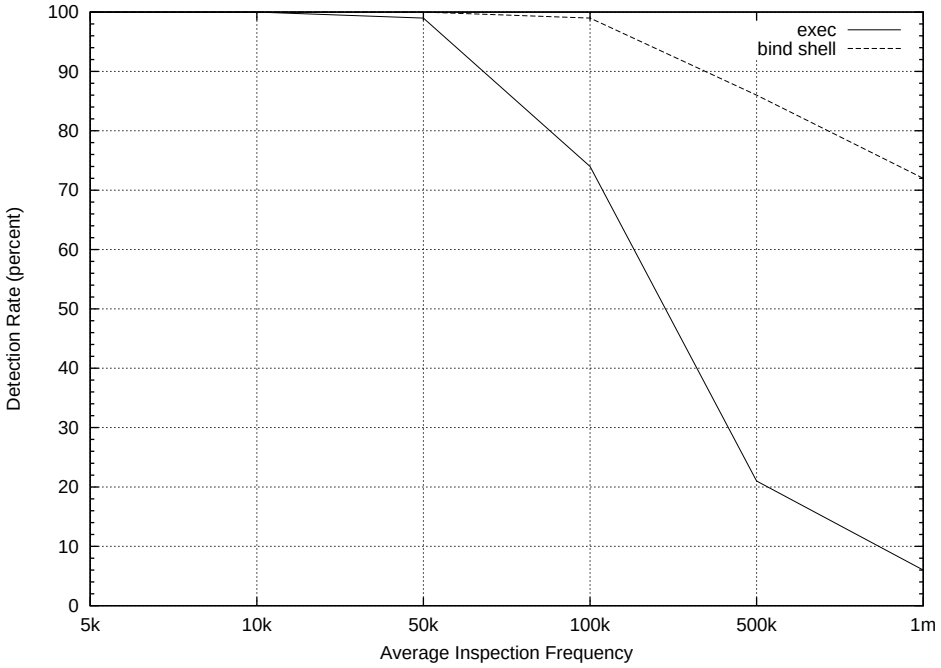

Figure 5.1: Detection rate for the two hypothetical attacks against Paint.NET, both exploiting the same hypothetical vulnerability, but with different shellcode. The first attack creates a process and then exit, while the second creates a command shell and binds it to a port. We have experimented WindRain2's capability to detect these two attacks when performing random inspection at different frequencies to show that it can detect injected code attacks in applications that execute data during normal operations.

is not allowed to generate code; thus any execution of the shellcode will be reported as an intrusion. In other word, the detection rate in Fig. 5.1 is entirely determined by WindRain2's ability to detect the execution of the shellcode.

5.4.3 Performance Overhead of the Enhancement to WindRain2

We start our discussion by reporting that our enhancement to WindRain2 for filtering false positives does not cause any noticeable slowdown to Microsoft Word, Visual Studio, PrimoPDF, Paint.NET and DAP (the 5 interactive / non-computation intensive applications among the 6 applications found to execute data in Sect. 5.1). While the tracing of writes to pages targeted by our policies will mean a significant cost for generating every instruction during runtime, this cost can be highly amortized by their repeated execution. In fact, many run-time systems that perform just-in-time compilation will only convert a piece of code from intermediate representation (e.g. Java bytecode) to native binary if that piece of code is found to be frequently executed. In other word, we believe the performance overhead caused by the tracing of all writes to pages that carry executable code will be very low even for applications that use JIT (that usually execute a lot of dynamically generated code).

We have also measured the performance overhead incurred by our enhanced version of WindRain2 on Java.exe. In particular, we've measured the performance of the JVMspec2008 benchmark while WindRain2 is running on the background, performing random inspection at various frequencies.

Our initial results show that the tracing of writes to pages holding dynamically generated code did cause Java to slow down very significantly. Further analysis shows that this is mainly caused by poorly specified policies concerning data execution in Java. In particular, we found that Java allocate multiple regions of virtual memory under the context targeted by our policies, but only one such region holds executable code, while all others contain only data. The tracing of writes to these data pages that are mistakenly targeted causes very frequent page fault, and thus incurs very high performance penalty. We have not found a better way to specify our policy, but instead, we have modified WindRain2 to identify these data pages based on their static start addresses, and repeated our experiments. The results of our second performance testing of Java are presented in Fig. 5.2 and 5.3.

From the results in Fig. 5.2 and 5.3, we see that the performance overhead incurred by WindRain2 on Java is quite acceptable once the average inspection frequency is at or below once every 500,000 instructions executed (with the exception of the benchmarks "serial" and "xml"). Also note that except for the two aforementioned benchmarks, the performance recorded when WindRain2 is performing inspection once every 1 million instructions executed is quite close to the baseline performance. This indicates that the performance overhead is mainly caused by the analysis performed at the inspection points (which include looking up the write records when WindRain2 detects data execution), while the cost of tracing write operations to the memory regions that hold dynamically generated code is not as high.

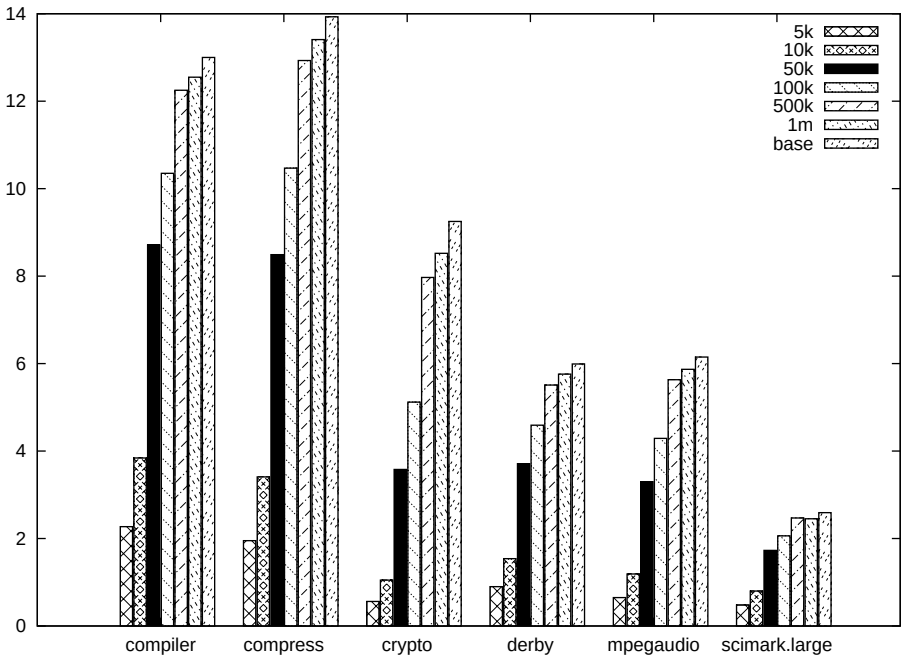

Figure 5.2: The performance of various benchmarks in the SPECjvm2008 suite when WindRain2 is tracing writes to memory regions that hold Java JIT code and perform random inspection at different average frequencies. Performance in SPECjvm2008 is measured in "operations/min"

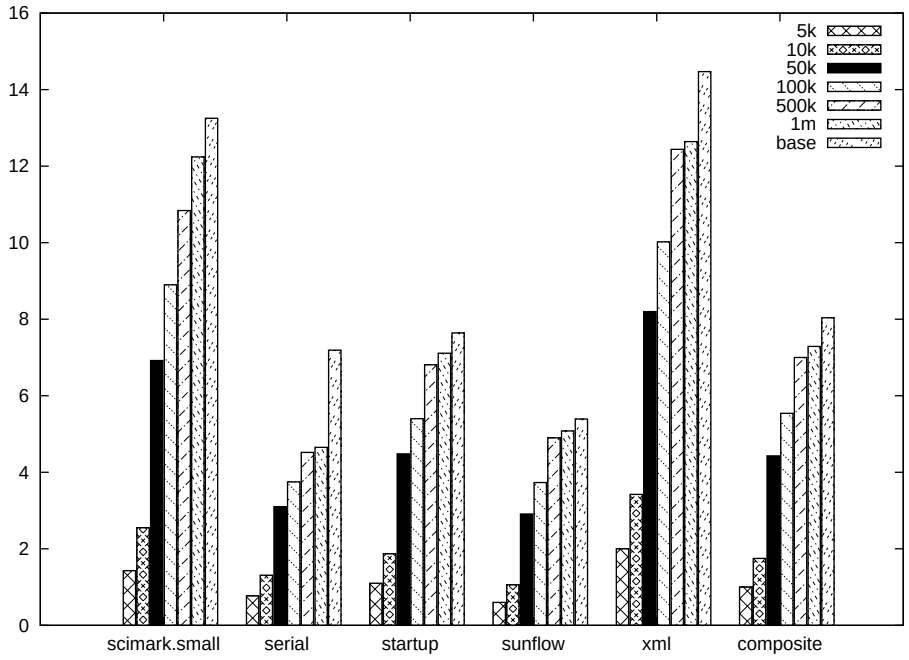

Figure 5.3: Continue from Fig. 5.2. The final set of data, labeled "composite" is the geometric mean of the performance of all the other 11 benchmarks.

5.5 Future Work

We believe the most likely cause of our problem with Java is that the memory regions that actually contain executable code and those that only carry data are allocated at different context, but our tracer program mistakenly reports them to be allocated under the same context due to the imperfection in our stack walking procedure (e.g. the call stack is "broken" because of functions that do not save their ebp registers). If this is the case, we only have to improve our stack walking procedure to solve our problem with Java (there are various documented techniques to retrieve return addresses from the stack while functions that do not save ebp registers are involved). We plan to find the real cause and a generic solution for our problem with Java in future work.

Another limitation of our approach to handle benign data execution lies in the need for per-application policies and the method we used to generate those policies. In particular, even though we've shown that in most cases, information collected by observing an application's behavior during start up and close down is sufficient for specifying reliable policies concerning its general data execution behavior, this approach for policy generation will not work for all applications. Furthermore, policies thus generated come with no guarantee at all; it is impossible to predict how well the information obtained by our brief "experiments" on an application can be generalized to cover its behavior while performing operations that we have not tested. The extremely high performance overhead incurred by our tracer program also makes it very difficult to perform more thorough testing of an application.

In our future work, we plan to tackle this problem in two directions:

1. We plan to optimize our tracer program to reduce its performance overhead; we believe this can be achieved by using more efficient data structures, as well as reducing the program's memory footprint.

2. We also plan to automate our "testing" process and improve its coverage by using automatic test case generation tools. If we can exercise every possible control flow path of an application while it is being observed by the tracer program, we will have a very good idea of the data execution behavior of the application under concern, and be able to generate policies to describe such behavior that come with strong guarantee. This is because we believe the occurrence of benign data execution behavior should be solely determined by the control path taken, but not the value of the data involved (in fact, we believe this to be a distinction between benign behaviors and vulnerabilities).

Finally, we are also interested in exploring the possibility of "borrowing" policies from one application to cover another. The general idea is that the policy describing the normal data execution behavior of Microsoft Word may also be applicable to PowerPoint and Excel; as another example, applications that use Java may also be able to share their policies. One possible way to implement this idea is to associate at least some of the policies with DLLs (i.e. an application that uses a particular DLL will be covered by the

policy associated with that DLL, but may also have policies regarding code-carrying memory regions that are specific to that application). We believe this approach will help to alleviate the burden of specifying a policy for every single application we want to cover with WindRain2.

Chapter 6

Collaborative Intrusion Prevention

In this chapter, we will present our framework for building collaborative intrusion prevention systems, as well as the implementation and evaluation of a prototype IPS that we've built based on this framework. We start our discussion by noting that even though the LAIDS/LIDS framework in Chapter 3 is designed under the traditional setting where the IPS employs honeypots to collect attack information, it provides a very good foundation for collaborative intrusion prevention. In particular, thanks to the self-contained analysis used in the LAIDS, we can easily distribute the task of monitoring different inspection points to different hosts under the collaboration while maintaining the effectiveness of the intrusion detection performed at each inspection point. All we need is a way to identify ALL possible inspection points, and to distribute the task of monitoring different inspection points to the different hosts under the collaboration. In the following, we will present one possible solution proposed in [79], and identify some shortcomings in the proposed scheme.

6.1 Related Work: Application Community

Like the CIP framework, the Application Community [79] distributes the work of detection and countermeasure generation over a community of hosts running the same application. The Application Community employs a static division of labor; all functions executed by the protected application are identified, and each participating host is assigned to monitor a fixed set of functions. We believe this static work allocation in [79] makes the Application Community scheme vulnerable to very similar problems as in traditional host-based IPSs, which are mentioned in Chapter 1; if the attackers can identify the hosts responsible for monitoring the vulnerable functions exploited by their attack, they can evade the Application Community scheme just as they evade the traditional host-based IPSs. By measuring the time it takes different hosts to process requests that utilize different functions, it is not impossible for the attacker to identify a superset of all hosts monitoring the target function.

The coverage of the protection provided by the scheme is also very much dependent on our ability to identify "all" functions under the protected application, which is not always an easy task in an environment that makes extensive use of DLLs (with different hosts running possibly different versions of the DLLs), and where applications using dynamically generated code are not uncommon. Also, since the work discovery/allocation is done on a per-application manner, the Application Community may have poor scalability; multiple communities have to be set up and managed, and each host has to join multiple communities so that all its applications are protected. The

mechanism for allocating task to different participants can also be an avenue of attack; this is especially true when tasks are allocated in a centralized manner.

The Application Community also appears to be a very altruistic scheme; with vulnerabilities usually found in the few rarely executed functions, we believe many hosts under the Application Community scheme are monitoring functions that will never be the target of any attack. In other word, very few hosts will directly benefit from the monitoring task performed for the Application Community scheme. We believe this nature of the collaboration provides very little incentive for hosts to participate.

Finally, the work in [79] did not provide any actual mechanism for monitoring each individual function of the protected application in a distributed manner. While the authors of [79] point to the literature for the underlying monitoring / detection mechanism, we note that many existing techniques are not suitable for the context-free setting in [79].

6.2 Using Random Inspection in the LAIDS/LIDS Framework

As an alternative to the static task allocation in [79], which can be seen as the root of many of the weaknesses in the Application Community idea, we propose to use random inspection as a means to probabilistically distribute the task of monitoring various possible inspection points to hosts in the collaboration, i.e. if each host in the collaboration performs random inspection independently, they'll effectively be monitoring a different subset of

all inspection points in the protected processes (and the subset of inspection points monitored by a hosts will change over time). In other word, if we assume all participating hosts perform random inspection at the same frequency, for any given inspection point involved in an attack, each attacked host will have a constant probability p of monitoring that point; i.e. a portion p of all the collaborating host will be monitoring the inspection point concerned, with the exact set of hosts performing such monitoring being "selected" randomly, on-the-fly, while the performance counter is filled with random values.

We believe the distribution of monitoring task through random inspection not only allows a very simple scheme of collaboration, but also makes the scheme very robust to failure of individual hosts; there is always a portion p of the remaining, uncompromised host that will monitor any given inspection point (including those that will lead to the detection of a new attack). Furthermore, the distribution scheme will automatically inherent the full coverage of the underlying random inspection; every possible inspection point in a protected application will have the same non-zero probability of being monitored. The by default system-wide coverage of random-inspection also means there is no need for separate installation/management for different applications we want to protect; once installed, the IPS will automatically cover all applications on the system. Finally, we note that the collaboration scheme based on random inspection provide hosts a lot of incentives to contribute to the collaboration; when a host performs high frequency random inspection, it is not only improving the chance of detecting any attack against itself, but also

increasing the probability that it will generate a countermeasure to help others stop the same attack.

However, we note that a straightforward application of random-inspection-based IDSs to the LAIDS/LIDS framework is going to result in a useless IPS. In particular, if we consider the analysis performed by the random-inspection-based IDS at each inspection point to be determining whether the current thread has been compromised, the countermeasures generated by the resulting IPS based on the LAIDS/LIDS framework will identify an inspection point within the attack shellcode. Such countermeasure will be easily defeated by attack polymorphism. In fact, even without attack polymorphism, the shellcode may appear at different locations in different attack instances (especially true for attacks that exploit heap buffer overflow vulnerabilities).

The above problem can be solved if we simply rephrase the analysis performed at each inspection point; instead of asking "has the current thread been compromised" (assuming a per-thread inspection), we consider the analysis to be determining "whether the current thread will be compromised in the next k instructions", where k is the number of instructions executed before the next inspection occurs. When we apply this paraphrased analysis to the LAIDS/LIDS framework, the countermeasures generated will convey the following information: "upon reaching execution point X, set the performance counter value to k so that an inspection will occur k instructions later". Also note that even though the above paraphrased analysis is not entirely self contained, it only requires the keeping track of a constant amount of information,

namely the identity of the previous inspection point, as well as the value k last assigned to the performance counter.

6.3 The Collaborative Intrusion Prevention Framework

By using the above paraphrasing technique and applying random-inspection-based IDSs to the LAIDS/LIDS framework, we obtain the Collaborative Intrusion Prevention (CIP) Framework. The operations of the hosts in IPSs built using the CIP framework can be summarized as follow:

1. Detection: In this normal state of operation, each participating host in the collaboration will try to detect new attacks by performing random inspection based intrusion detection at a low inspection frequency of its own choosing. After each inspection, if no intrusion is detected, the identity of the inspection point, as well as the new random value in the performance counter will be recorded; separate records will be kept for each thread in the system.

2. Countermeasure Generation: Once an intrusion is detected, the record for the previous inspection point and the saved performance counter value of the thread involved are retrieved. These are basically all the information needed for a countermeasure against the detected attack. However, the information retrieved may be refined to make the countermeasure more portable among hosts. For example, the virtual address of the inspection point involved may be expressed as the name of the

containing module and the offset from the beginning of the module. After all the necessary refinement, the countermeasure will be distributed to all other hosts.

3. Patching: Upon receiving a countermeasure from another host, the inspection point identified will be extracted, and a breakpoint will be inserted at the corresponding instruction so that an exception will be generated every time it is executed. A record will also be established to associate the marked instruction with the performance counter value, k[1], given in the countermeasure.

4. Stopping Attacks: If any instruction marked in the patching stage is executed, the CPU will raise a breakpoint exception. When handling the exception, the performance counter for the interrupted thread will be set to the value k (the value associated with the corresponding instruction), so an inspection will occur after k user space instructions are executed in that thread. Normal low frequency inspection of the detection phase will resume if no intrusion is detected after the next inspection. In addition to configuring when the next inspection should occur, the interrupted thread can also take measures to facilitate the recovery should any attack be detected. For example, modifications to critical resources can be delayed or redirected, and only be committed if no intrusion is detected

[1]in the following discussion, we will refer to the performance counter value associated with a countermeasure as "k"

at the next inspection point (in [35], these are called cordoning-in-space and cordoning-in-time respectively).

6.3.1 Premature Firing

When we try to implement our prototype CIP system by applying WindRain2 to the above framework, we find that countermeasures generated by our IPS can identify frequently executed instructions, and this can lead to two related problems:

1. inserting breakpoints at the frequently executed instructions identified by the countermeasures can result in a lot of breakpoint exceptions, which will in turn significantly slow down the protected system (as we can see in Sect. 4.7, even an interrupt once every 5000 instructions executed in user space can lead to more than 100% slowdown).

2. if the instruction advertised by a countermeasure is executed multiple times in the processing of a malicious input, there is no way we can tell when we should set the performance counter to k and schedule the inspection intended by the countermeasure; if we set the counter to k too early, we may perform the inspection before the corresponding thread enters an illegal state; if we start too late, we may perform the inspection after the shellcode has finished execution and killed the compromised thread / process; either way, we miss the attack.

Since we believe the first problem is much more serious than the second (it's better to build an IPS that sometimes misses attacks than one that slows down the system by 100%), we've modified our breakpoint handling process as follow: when an instruction advertised by some countermeasure is executed and generates a breakpoint exception, we remove the breakpoint at that particular instruction. The breakpoint will be restored after k instructions have been executed in the offending thread. This way, we can keep the performance overhead caused by the breakpoint exceptions to a minimal (unless we have applied a very large number of countermeasures, or have accepted countermeasures with extremely small k values).

As for the second problem of multiple executions of the same instruction advertised by a countermeasure during the processing of malicious input, we have not yet identified a good solution that does not incur significant overhead. Nonetheless, as we'll show in the next section, one very effective remedy to this problem is to have multiple countermeasures for the same attack. Another remedy that we have implemented in our prototype is as follow: instead of executing one inspection exactly at the k-th instruction executed after the breakpoint exception, we perform a number of inspections at high frequency at the end of the k-instruction period (for our implementation, we have 5 inspections at an average frequency of once every 100,000 instructions executed).

We believe the above design will allow us to detect the attack even if we start counting the k instructions slightly too early, or slightly too late. This is because under WindRain2, attacks tend to put the compromised process /

thread in intermittent long periods of illegal state execution; in other word, the shellcode will stay in an illegal state for many instructions, leave the illegal state for a while, and then return to the illegal state to execute another long sequence of instructions. This property allows us to make slight error in when we start counting the k instructions as intended by the countermeasure. The reason of performing a number of inspections at high frequency towards the end of the k-instruction period is that if the inspection at the k-th instruction occurs while the shellcode has enter a normal state, we maybe able to catch it in an illegal state with the other inspections before the last one. The above design also made the implicit assumption that the detection which leads to the generation of the countermeasure did not occur at the first or the last instruction executed in illegal state; instead, it assumes the detection occurred somewhere in the middle of the execution of the shellcode. Since we expect attacks to put the compromised process / thread in illegal states for at least tens of thousands of instructions, this assumption should be true most of the time. As such, we can start the counting of the k instructions slightly early, and still expect the k-th instruction to be executed after the shellcode has started. Similarly, if we start counting slightly too late, it is still very likely that some of the inspections before the end of the k-instruction period will occur before the shellcode finishes its execution.

6.3.2 Evaluating our Prototype

We've implemented our prototype collaborative intrusion prevention system based on the above modified CIP framework, using WindRain2 as the underlying IDS, and our prototype works for Windows XP with no service pack or with service pack 2. Before we present the results of our experiments with the prototype, let's consider how low we can make the performance overhead incurred by performing random inspection during the normal "detection" state. In particular, we have measured the performance overhead incurred on gzip, the javascript benchmark and the css benchmark while our IPS is performing random inspection at the frequencies of once every 5 million and 10 million instructions; we run each benchmark ten times and use the average time for the ten runs in computing the performance overhead. For all the experiments presented in this chapter, the IPS (or the underlying WindRain2) is configured to retrieve 20 return addresses from the stack at every inspection point. Also, for experiments that measure the performance overhead incurred by our IPS, we report a "0%" performance overhead when the average runtime of the benchmark at the studied configuration is lower than the base time obtained when our IPS is not running on the background. Our results are presented in Table 6.1.

As we can see in Table 6.1, our prototype IPS allows collaborating host to participate while incurring a very low performance overhead; this is achieved through decreasing the average frequency at which inspection occurs. Now let's consider how well countermeasures generated by hosts performing

	5M	10M
gzip	2.66%	1.6%
Javascript benchmark	0%	0%
CSS benchmark	0%	0%

Table 6.1: Performance overhead incurred on three different benchmarks when WindRain2 is running on the background performing random inspection at an average frequency of once every 5 million and 10 million instructions executed respectively.

random inspections at such low frequencies are in detecting attacks. We have experimented with attacks against three actual vulnerabilities on Windows applications, namely: Apache Win32 Chunked Encoding, Internet Explorer createTextRange() (ms06-013), and Internet Explorer Daxctle.OCX KeyFrame method (ms06-067). For every vulnerability, we use metasploit to construct our attacks. As for the attack payload, we use the same two shellcode tested in Sect. 4.6. We start by measuring the detection rate of our prototype IPS for the six different (vulnerability, shellcode) combinations while the underlying WindRain2 is performing random inspection at the low frequencies of once every 5 million and 10 million instructions executed respectively. The results of this experiment are presented in Table 6.2.

From the results in Table 6.2, WindRain2 has at least 1% detection rate for each of the six tested attacks, even when random inspection is performed at a frequency of once every 10 million instructions executed, and the attack employs an unrealistically simple shellcode. This means fewer than 100 hosts in the collaboration will be compromised before an attack is first detected

	exec		bindshell	
	5M	10M	5M	10M
Apache	2%	2%	60%	50%
Ms06-013	12%	4%	98%	92%
Ms06-067	2.4%	1.2%	93%	91%

Table 6.2: Probability for an individual host to detect three different attacks while operating in the detection phase under the CIP framework, performing random inspection with frequencies once every 5 million and 10 million instructions.

and a countermeasure against the attack becomes available [2]. Similarly, we expect at least 5 countermeasures will be generated when 500 hosts in the collaboration has been attacked. We also note that the significant difference in detection rate for the attacks against different vulnerabilities can be caused by the specific exploitation technique employed (e.g. heap spray, which may require a long sled to be executed before the real payload starts, vs. simple stack overflow with register spring, which does not need a sled at all).

After establishing some estimate of the size of the collaboration required, we use our prototype to generate 5 countermeasures for each of the tested vulnerability, assuming the first, simpler shellcode is employed. As before, we repeat our experiments for the settings where the underlying WindRain2 is performing inspection at an average frequency of once every 5 million and 10 million instructions executed. For each countermeasure, we evaluate its detec-

[2]This number is obtained by considering the detection of an attack by a single host to be a Bernoulli trial with success probability of at least 0.01, thus the expected number of success after 100 trial is at least 1

	gzip overhead	javascript benchmark overhead	CSS benchmark overhead
# 1	1.78%	12.23%	7.5%
# 2	1.34%	0%	0.33%
# 3	2.31%	0.35%	2.17%
# 4	2.04%	2.55%	0.83%
# 5	2.57%	0.35%	0%
all	2.74%	11.24%	6.5%
all_below_5	2.04%	0%	0%

Table 6.3: Performance overhead incurred by the 5 countermeasures for the attack against Apache server, generated when hosts are performing random inspection once every 5 million instructions executed.

tion rate against the two versions of the corresponding attack, each exploiting the same vulnerability, but with different shellcode. We repeat our experiment 100 times for each attack (with the IPS performing a "background" random inspection for the detection phase once every 10 million instructions executed on average). We also measure the performance overhead incurred on gzip, the Javascript benchmark and the CSS benchmark by applying the different countermeasures. Finally, for each set of 5 countermeasures generated, we measure the detection rate and performance overhead when all 5 countermeasures are applied, as well as when all countermeasures that incur a performance overhead of less than 5% on the three benchmarks are applied. The results of our experiments are presented in Fig. 6.3 to 6.14.

The results in Table 6.3 to 6.14 show that the performance impact of countermeasures generated by our prototype CIP can show very significant

	exec	bindshell
# 1	100%	(99%, 0%)
# 2	99%	(99%, 0%)
# 3	99%	(94%, 6%)
# 4	0%	(99%, 1%)
# 5	94%	(100%, 0%)
all	78%	(99%, 0%)
all_below_5	99%	(99%, 0%)

Table 6.4: Detection rate of the 5 countermeasures for the attack against Apache server, generated when hosts are performing random inspection once every 5 million instructions executed.

	gzip overhead	javascript benchmark overhead	CSS benchmark overhead
# 1	1.95%	0%	0%
# 2	2.04%	0%	0%
# 3	1.6%	1.56%	1%
# 4	2.22%	0%	1%
# 5	4.23%	0%	1.33%
all	2.74%	0.21%	0%
all_below_5	2.74%	0.21%	0%

Table 6.5: Performance overhead incurred by the 5 countermeasures for the attack against Apache server, generated when hosts are performing random inspection once every 10 million instructions executed.

	exec	bindshell
# 1	70%	(98%, 1%)
# 2	67%	(95%, 4%)
# 3	66%	(88%, 10%)
# 4	94%	(99%, 2%)
# 5	67%	(98%, 0%)
all	99%	(99%, 0%)
all_below_5	99%	(99%, 0%)

Table 6.6: Detection rate of the 5 countermeasures for the attack against Apache server, generated when hosts are performing random inspection once every 10 million instructions executed.

	gzip overhead	javascript benchmark overhead	CSS benchmark overhead
# 1	1.52%	0%	1.33%
# 2	3%	5.09%	0%
# 3	2.04%	0.42%	0%
# 4	2.22%	3.75%	0%
# 5	1.78%	7.5%	0%
all	1.52%	13.08%	0%
all_below_5	1.6%	9.26%	0%

Table 6.7: Performance overhead incurred by the 5 countermeasures for the attack targeting the ms06-013 vulnerability, generated when hosts are performing random inspection once every 5 million instructions executed.

	exec	bindshell
# 1	24%	(91%, 9%)
# 2	85%	(100%, 0%)
# 3	69%	(91%, 9%)
# 4	35%	(100%, 0%)
# 5	62%	(69%, 30%)
all	80%	(100%, 0%)
all_below_5	88%	(99%, 1%)

Table 6.8: Detection rate of the 5 countermeasures for the attack against ms06-013 vulnerability, generated when hosts are performing random inspection once every 5 million instructions executed.

	gzip overhead	javascript benchmark overhead	CSS benchmark overhead
# 1	2.74%	9.34%	0.33%
# 2	3.97%	2.05%	0%
# 3	0.99%	2.12%	14.67%
# 4	2.74%	4.17%	0%
# 5	1.69%	8.06%	0.33%
all	2.04%	10.8%	12.17%
all_below_5	1.95%	3.11%	0%

Table 6.9: Performance overhead incurred by the 5 countermeasures for the attack targeting the ms06-013 vulnerability, generated when hosts are performing random inspection once every 10 million instructions executed.

	exec	bindshell
# 1	78%	(93%, 7%)
# 2	55%	(99%, 1%)
# 3	88%	(96%, 4%)
# 4	54%	(98%, 2%)
# 5	77%	(100%, 0%)
all	95%	(100%, 0%)
all_below_5	66%	(100%, 0%)

Table 6.10: Detection rate of the 5 countermeasures for the attack against the ms06-013 vulnerability, generated when hosts are performing random inspection once every 10 million instructions executed.

	gzip overhead	javascript benchmark overhead	CSS benchmark overhead
# 1	2.57%	0%	0%
# 2	5.9%	6.44%	1%
# 3	1.6%	15.13%	0%
# 4	1.78%	7.64%	0%
# 5	2.48%	0%	1.33%
all	5.46%	16.05%	0.167%
all_below_5	2.31%	1.91%	0%

Table 6.11: Performance overhead incurred by the 5 countermeasures for the attack targeting the ms06-067 vulnerability, generated when hosts are performing random inspection once every 5 million instructions executed.

	exec	bindshell
# 1	0%	(0%, 95%)
# 2	92%	(63%, 37%)
# 3	98%	(92%, 8%)
# 4	94%	(89%, 10%)
# 5	100%	(100%, 0%)
all	99%	(98%, 2%)
all_below_5	99%	(98%, 0%)

Table 6.12: Detection rate of the 5 countermeasures for the attack against the ms06-067 vulnerability, generated when hosts are performing random inspection once every 5 million instructions executed.

	gzip overhead	javascript benchmark overhead	CSS benchmark overhead
# 1	1.34%	0%	0%
# 2	3.71%	13.51%	15.17%
# 3	3.62%	0%	2.17%
# 4	2.22%	12.59%	0%
# 5	1.87%	4.81%	8.33%
all	3.01%	15.13%	13%
all_below_5	2.48%	3.68%	0%

Table 6.13: Performance overhead incurred by the 5 countermeasures for the attack targeting the ms06-067 vulnerability, generated when hosts are performing random inspection once every 10 million instructions executed.

	exec	bindshell
# 1	99%	(100%, 0%)
# 2	98%	(100%, 0%)
# 3	44%	(30%, 16%)
# 4	99%	(95%, 3%)
# 5	85%	(98%, 2%)
all	98%	(100%, 0%)
all_below_5	100%	(99%, 0%)

Table 6.14: Detection rate of the 5 countermeasures for the attack against the ms06-067 vulnerability, generated when hosts are performing random inspection once every 10 million instructions executed.

variation. Furthermore, there appears no correlation between the effectiveness and the performance overhead caused by a countermeasure; i.e. we can have countermeasures with almost 100% detection rate for the corresponding attack with the first shellcode, and yet incur only 2% of overhead on our benchmarks. We believe such lack of obvious trend in our experimental results is caused by the non-deterministic nature of our CIP framework. In other word, the instruction associated with a countermeasure generated by our IPS can be one that is frequently executed by many different applications (e.g. some popular function in ntdll.dll), or it can be very specific to the vulnerability being exploited.

As for the detection rate, while there are significant fluctuations against attacks that employ the first shellcode, the countermeasures can reliably detect the corresponding attack when the second shellcode (bindshell) is used; 25 out of 30 countermeasures achieve a 99% detection rate against attacks that

employ the second shellcode, with the host performing background inspection once every 10 million instructions (as opposed to the 92% maximum baseline detection rate). Furthermore, in 12 out of the 30 cases, 99% of the attacks are detected within the k-instruction period, and there is at least one such countermeasure in each of the 6 settings. In other word, at least one of the 5 countermeasures in each set will allow a 99% detection rate while the host performs no background random inspection at all.

We would also like to point out that while our countermeasures do not always achieve a very high detection rate when the first shellcode is used in the attack, they do lead to significant improvement over the base detection rate. In fact, for such attacks, countermeasures generated by our IPS can be seen as means to improve the trade off between detection rate and the performance overhead incurred by the underlying WindRain2. In particular, we observe that while the performance overhead incurred by various countermeasures is at a level observed when WindRain2 is operating at an inspection frequency of once every 100,000 to 500,000 instructions executed, the detection rate is usually at the level observed at a much higher inspection frequency.

Finally, the results in Table 6.3 to 6.14 show that applying multiple countermeasures is the best defense against the fluctuation in quality of individual countermeasures. Even for the worst case scenario where the first simple shellcode is used, the application of all 5 countermeasures will result in a higher than 98% detection rate in 4 out of 6 sets of countermeasures. The results of applying all the countermeasures with a performance overhead of

less than 5% is especially encouraging; the resulting performance overhead for all three benchmarks is lower than 5% except for one case, while the detection rate is similar to that of applying all 5 countermeasures.

We'll conclude our evaluation of the prototype collaborative intrusion prevention system by stating that even countermeasures generated do not always have a high detection rate, they usually significantly increase a host's chance of detecting an attack, and can be very useful in stopping or limiting the damage caused by an outbreak like SQLSlammer. In other word, even though offering only very crude defense, we believe IPSs based on our CIP framework can be good complements to more accurate but much slower techniques that exist in the literature.

6.4 Future Work

From the lack of correlation between a countermeasure's detection rate and the performance overhead observed in the previous section, as well as our results of applying all countermeasures with a less than 5% performance overhead, we believe a system that allows hosts to predict the performance overhead incurred by a received countermeasure will be of great value. With such system, hosts can choose to reject countermeasures which are expected to have significant performance impact, and wait for better quality ones (i.e. those with high detection rate and low overhead). We believe the risk in such wait is quite bearable for many hosts, since our results in the previous section show that in most cases, a good quality countermeasure will become available

before 500 hosts are compromised. Thus, in our future work, we plan to build a profiling system on top of random inspection, and predict the performance overhead incurred by inserting a breakpoint at the instruction advertised by a countermeasure. One challenge we may face in building such system is how to build a compact, yet accurate profile for the frequency of execution of every instruction in every library on a host.

We also realize that in our implementation of the prototype collaborative intrusion prevention system, concurrent firing of breakpoints inserted by various countermeasures are handled in a very simple manner; in particular, if the instruction identified by countermeasure A triggers a breakpoint exception, we will set the performance counter according to the information in A, even if another countermeasure B has just fired and the k-instruction period associated with the firing of B is not over yet. The adverse result of such naive handling of multiple firing can be seen in the results presented in Table 6.3 to 6.14; while applying multiple countermeasures usually lead to a performance overhead that is similar to that of the single countermeasure with the highest overhead, the resulting detection rate is not always better than the best individual countermeasure. One possible solution to this problem is to always choose an earlier starting point and a later ending point for the high frequency inspection around the end of the k-instruction period after the firing of the breakpoints. We believe there are also many other possible solutions with different tradeoff between detection rate and performance overhead. In our future work, we plan to explore the different possible policies for handling

multiple firing and identify one with the best tradeoff.

We are also interested in exploring other solutions for handling the premature firing problem discussed in Sect. 6.3.1. In particular, in addition to the identity of the instruction where the inspection point preceding the actual detection occurs, we would like to include the context of this inspection point (i.e. return addresses retrieved from the stack). With such information, we can determine whether an execution of the instruction associated with a countermeasure is a premature firing, and possibly handle the breakpoint exception differently (e.g. if it is a premature firing, we may choose not to remove the breakpoint until a number of premature firing is observed, or we may remove the breakpoint and set the performance counter to a smaller value than suggested by the countermeasure, and restore the breakpoint at the next inspection).

Chapter 7

Conclusions

In this dissertation, we proposed the collaborative intrusion prevention framework in which multiple hosts collaborate to detect and collect information about new attacks and generate countermeasures to help other hosts defend against attacks detected. We motivated our work by pointing out the weaknesses in the current practice of using honeypots to collect attack information in host-based intrusion prevention systems (IPSs):

1. the honeypot becomes the single point of failure in the IPS; with existing techniques that allow attackers to identify the IP addresses of our honeypots, IPSs that use honeypot to collect attack information can be easily blinded,

2. the use of honeypots also creates scalability problems; in order to protect all the hosts in a network, a large number of honeypots will have to be set up and managed for the great variety of OSs and applications used.

3. the passive nature of honeypots also makes them unsuitable for collecting information about attacks that require some user actions (e.g. exploits of vulnerability in web browsers usually require the user to visit some contaminated webpages).

We also defended our focus on host-based IPS by demonstrating the difficulties facing purely network-based IPSs which do not collect any information on the victim hosts. In particular, we showed that even the most advanced network-based IPS can be tricked into believing that benign network traffic of the attacker's choosing to be some sort of new attack and block them at the perimeter defense, thus effectively creating a DoS against the protected network. We call this the allergy attack.

After providing the motivation of our work, we presented two ideas that form the foundation of the proposed framework for collaborative intrusion prevention, namely the LAIDS/LIDS framework for building host-based IPS and the random-inspection-based IDS. The LAIDS/LIDS framework identifies a class of IDS that are very suitable for detecting new attacks and collecting attack information in an IPS; we call this class of IDSs the LAIDS (lazy-able IDS). Once an IDS in this class is identified, we can easily construct an IPS using this IDS. In particular, when the LAIDS detects a new attack, it will simply broadcast the identity of the point where the attack is detected, so that hosts protected by the IPSs can perform the same analysis that leads to the detection at the first place. Thanks to the properties of the intrusion detection performed by LAIDSs, such mimicking of the first detection will allow protected host to stop future instances of the attack with a very small performance overhead. To demonstrate the usefulness of the LAIDS/LIDS framework, we have identified one example of IDS in the LAIDS class, Program Shepherding and presented an IPS built by putting Program Shepherding into

the LAIDS/LIDS framework. Our evaluation shows that the IPS thus constructed can indeed generate useful countermeasures against popular attacks; furthermore, we've shown that the cost of detecting attacks using information in these countermeasures is very low (below 3%).

As for random-inspection-based intrusion detection, it was first proposed as an alternative to the popular approach of monitoring processes at the system call interface, which is plagued by mimicry attack. The main idea behind random-inspection-based intrusion detection is to randomly and preemptively stop a monitored process to check whether it has been compromised. We believe the non-deterministic and preemptive nature of such checking makes it more difficult for attackers to evade detection. Since they cannot predict when the next checking will occur, there is no way they can guarantee that the state of the compromised process will appear normal at the next checking. The preemptive nature of the checking also means the attackers cannot prevent the checking from occurring by avoiding certain operations. We've also presented our prototype random-inspection-based IDS, WindRain2. Our experiments show that WindRain2 can detect any realistic attack with 100% accuracy while maintaining a performance overhead of less than 10%.

We have also studied the false positive behavior of WindRain2, and showed that it is not uncommon for benign applications to execute code in data space and lead to false positives in WindRain2. Based on our study of how benign applications execute data, we have presented an enhancement to

WindRain2 that allows it to distinguish between benign data execution and actual injected code attack. Our experiments show that this enhancement allows WindRain2 to filter out the false alarms from many benign applications without affecting its ability to detect real attacks against these applications.

Finally, by combining both the LAIDS/LIDS framework and the idea of random-inspection-based IDS, we proposed our framework for building collaborative intrusion prevention systems, and presented one example of IPSs that realizes this framework. Our prototype IPS is based on WindRain2. Our evaluation shows that our prototype IPS allows hosts to collaborate in intrusion prevention at a very low cost, and the countermeasures generated by such collaboration are very effective against the most common attacks; even in the worst case scenario where the attack shellcode is unrealistically simple, the countermeasures generated can still significantly improve a host's chance of detecting the attack while incurring a very small performance overhead. Furthermore, we've shown that only a few hundred of hosts in the collaboration will be compromised before useful countermeasures against a new attack become available.

Appendices

Appendix A

Autograph

In this appendix, we will present some background information about Autograph, the network-based IPS studied in the first part of Chapter 2.

Autograph is a string-matching system that generates worm signatures by monitoring traffic crossing an edge network's DMZ. Since the Autograph prototype available to us only handles TCP packets, we assume all traffic to be under the TCP protocol. Signatures generated by Autograph are destination port specific, i.e. only traffic destined to the corresponding port will be matched against a signature.

Autograph processes traffic in two stages. In the first stage, Autograph identifies scanners by recording IP addresses that made more than *s_thresh* unsuccessful connection attempts to the protected network. A connection attempt is considered unsuccessful if it times out without any reply received, or it got reset before completing the TCP handshake. In addition to the IP address, Autograph will also record the destination ports targeted by all the failed connections from a scanner. Afterwards, all the TCP packets from successful connections originating from a scanner address and destined to a recorded port will undergo flow reassembly. The resulting suspicious flows

will be recorded in a suspicious pool. With enough flows in the pool that are destined to the same port, Autograph will start the next stage of processing: signature generation.

In the signature generation stage, Autograph will divide the suspicious flows into content blocks, and find the set of most prevalent blocks. The process is greedy, and the block with highest prevalence will be picked first. Autograph will keep adding blocks to the set until a pre-configured portion of suspicious flows contain one or more blocks from the set. Signatures will then be generated for each block in the set, with the entire block being the byte sequence that will be matched against future traffic destined to the port for which signature generation is invoked.

For dividing flows into content blocks, Autograph employs the COntent-based Payload Partitioning (COPP) technique. The COPP partitions suspicious flows into non-overlapping, variable-length blocks by computing the Rabin fingerprint of every 2-byte subsequence in the flow, starting from its beginning. The 2-byte subsequence marks the end of a content block if it matches B, i.e. its fingerprint r satisfied the equation $r = B \ (mod \ a)$, where B is a predetermined breakmark, a is a configurable parameter that controls the average block size. Due to the content based nature of COPP, a similar set of blocks will be generated even if bytes are added to or deleted from the worm payloads. This helps Autograph to generate signatures that filter different instances of a polymorphic worm.

To avoid overly specific or overly general signatures, Autograph bounds

the size of content blocks generated between m bytes and M bytes (with m and M configurable). In other word, Autograph will not end a content block at a 2-byte subsequence that matches B if that results in a block shorter than m bytes. Instead, Autograph will search for the next matching 2-byte subsequence. Similarly, any content block that reaches M bytes long will be terminated. Autograph also avoids using content blocks in flows that originates from fewer than a configurable *source_count* number of sources for signatures. This prevents generating signatures for normal traffic from misconfigured, but benign hosts. Finally, Autograph employs a blacklisting mechanism which prevents subsequences of any blacklisted byte sequences from being used as signatures. In [42], the blacklist is generated in a training period where all signatures generated are manually checked for false positives. Signatures generated in this period which are deemed to match normal traffic will be added to the blacklist. This prevents generating signatures for normal traffic that Autograph normally misclassifies.

Appendix B

Bypassing Blacklisting in Autograph

In this appendix, we will present our attack against Autograph under the worst case scenario where all content blocks from the target request are blacklisted in the training phase. We note that such blacklisting will thwart the simple attack described in 2. However, this obstacle is circumventable.

B.1 Design of the Attack

To begin with, we observe that during the training phase, our target request will always be partitioned in its entirety, and thus always result in the same set of content blocks. In other word, new content blocks that are not blacklisted may be generated if a fragmented target request is partitioned by COPP. For example, let the target request be the byte sequence $b_0b_1b_2...b_{i-2}b_{i-1}b_ib_{i+1}...b_n$, with $n - i > m$ and $i - 1 > m$. Suppose b_i is the last byte of the first content block generated by COPP. If the byte sequence $b_{i-2}b_{i-1}b_ib_{i+1}...b_n$ is presented to COPP, a content block starting with b_{i-2} will be generated. Since Autograph is not producing blocks of less than m bytes, the block will continue at b_i. More importantly, though this new content block contains bytes from two blacklisted blocks (the ones starting from

b_1 and b_{i+1} respectively), it is a substring to neither. As a result, the allergy attack against the target request will be successful if we use $b_{i-2}b_{i-1}b_i...b_n$ in our attack packets. Another point worth noting is that the above strategy will remain effective even if requests similar to our target are also blacklisted. This is because these similar requests will result in mostly the same set of content blocks being blacklisted, due to the content-based nature of COPP.

Without knowing the configurations of the COPP (e.g. m, a, and B), we do not know where the boundaries of content blocks lie when the target request is partitioned in the training phase. In other word, we do not know exactly which fragmented target request will result in content blocks that overlap two adjacent blocks in the original partition. Nonetheless, we can approximate the above strategy by using random, fixed-length subsequences of the target request. With sufficient trails, some of these subsequences will result in new, non-blacklisted content blocks, and achieve our goal. Note that by using fixed-length subsequences with random starting points instead of random suffixes of the target request, we vary the last bytes of the various suspicious flows partitioned, and improve our chance of success. This is because COPP usually generates a content block with the last m bytes of the flow partitioned (unless the last content block ends exactly at the end of the flow).

Based on the above observations, the concrete design of our attack is as follow: as in Sect. 2.2.2, we start by having all our drones classified as scanners by Autograph; we will then divide the rest of our attack into different rounds, and in each round, all drones will pick the same NUM_SEQ

random subsequences of length SEQ_LEN from the target request. This can be achieved by synchronizing all drones to start the round at roughly the same time (with synchronization error of up to a few minutes), and use the same seed for the same random number generator. Each drone will then send each chosen subsequence to the target network over NUM_REP connections destined at the target port. The next round of attack will then begin t minutes after the completion of the previous round. This is to make sure that the suspicious flows from the previous round have expired (i.e. removed from the suspicious pool). Obviously, a larger NUM_SEQ will reduce the number of rounds needed to achieve a successful attack. For our experiments, we use two drones, with NUM_SEQ=30, and NUM_REP=50. We emphasize that we use such a small number of drones only to ease our experiments, we don't see any technical difficulties in a 10-fold or even a 100-fold increase in the number of drones. As for the value of SEQ_LEN, a proper choice of SEQ_LEN will significantly improve our chance of success. However, since the best value of SEQ_LEN depends on the unknown parameters a and m of the attacked Autograph system, a trial-and-error process over multiple rounds is necessary. Nonetheless, our experiments show that for all reasonable values of a and m, it is very likely for the attack to succeed in just one round with SEQ_LEN being 80 to 160. Finally, the choice of t will depend on the *t_thresh* parameter of Autograph. With the default value of *t_thresh* being 30 mins, we can safely assume the actual value being less than 90 minutes, since a *t_thresh* value higher than 90 minutes can lead to a prohibitively large suspicious pool (a

similar argument appears in [80]).

B.2 Experiments

To demonstrate the effectiveness of our attack, we have tested it against Autograph for the HTTP requests to the following three webpages:

1. http://www.cs.utexas.edu
2. http://www.cs.utexas.edu/users/mok
3. http://www.cs.utexas.edu/users/mok/cs372/Fall05/projects/lab1/index.html

We generate the target requests with internet explorer (IE). For each target request, we pick a set of 10 seeds for generating the random subsequences in 10 different rounds of attack. The same 10 seeds are then used for the experiments with SEQ_LEN at 40, 80, 120 and 160. This allows us to compare the effectiveness of our attack at different SEQ_LEN without being affected by the randomness in the seeds used. The effectiveness of our attack is measured by the average number of distinct signatures generated for each of the 10 rounds under the same SEQ_LEN[1]. Finally, to test how the different configurations of Autograph affect the effectiveness of our attacks, we repeat our experiments at different values of a (with a=16, 32, 64, and 128) and m

[1]Even though any single signature generated will completely block out the target request, we believe the number of signatures generated will reflect the robustness of our attacks for different targets.

(with m=16, 32, and 64), which is basically the range for a and m tested in [42]. For the other parameters of Autograph, we simply use the default values. We choose to focus our experiments on a and m because these two parameters have the most impact on the success of our experiments.

A point worth noting is that our attack is specific to the web browser used. In other word, the signatures generated will mostly filter out requests from IE only. Other web browsers (e.g. Mozilla) are thus unaffected. Nonetheless, a determined attacker can launch a separate attack for each popular web browser. Since the market is mainly dominated by a few web browsers, we believe this is not a major undertaking.

Instead of installing Autograph to monitor real traffic crossing an edge network's DMZ, we choose to perform our experiments offline by feeding Autograph with traffic traces captured separately at the two drones used. This will expose Autograph to exactly the attack traffic originating from the drones, as well as the reply from the attacked network that Autograph is supposed to be protecting, while ignoring all other traffic to/from the drones. As a result, we are presenting to Autograph only the "slice" of traffic that is relevant to our attack. This approach greatly simplifies our work, and allows us to test the same attack traffic under different configurations of Autograph.

A disadvantage of the above approach is that it prevents us from studying the effect of the background noise to our attack. For background noise, we are referring to the scanning activities that happen constantly on the internet, as well as small-scale worm outbreaks over the world. The effect of

these events is mainly to populate the suspicious pool of Autograph with flows other than those from our attack. Nonetheless, we believe we can easily make content blocks from these flows an insignificant portion of the suspicious pool. By increasing NUM_REP to 200 and having 20 drones instead of 2, we can almost guarantee that content blocks from our attack packets will be sufficiently prevalent to be used as new signatures (each will have 4000 copies in the suspicious pool). Furthermore, even with the higher NUM_REP and increased number of drones, we believe our attack is still entirely feasible.

Finally, to validate the claim that our attack will remain effective even if the target requests and some related requests are blacklisted during the training phase, we populate the blacklist with all content blocks from the three target requests as well as the requests for the following 5 related pages:

1. http://www.cs.utexas.edu/users
2. http://www.cs.utexas.edu/users/mok/cs372
3. http://www.cs.utexas.edu/users/mok/cs372/Fall05
4. http://www.cs.utexas.edu/users/mok/cs372/Fall05/projects
5. http://www.cs.utexas.edu/users/mok/cs372/Fall05/projects/lab1

A separate blacklist is generated for each tested Autograph configuration by using the entire request to be blacklisted (instead of its subsequences) in the attack described above. Every time Autograph generates a signature

for our "attack traffic", we add it to the blacklist. We repeat the "attack" until no more signature is generated (i.e. all content blocks are blacklisted).

After describing the experimental setup, we will present our results on the first target request in Fig. B.1. The results for the other two targets are very similar to those of the first one, and are therefore elided for brevity.

Our experiments show that the attack presented in Sect. B.1 is very effective for all three target requests. At all combinations of SEQ_LEN, a and m where $SEQ\_LEN \geq m$, at least 8 out of the 10 rounds of our attack successfully induced Autograph into generating one or more signatures. Thus, we are confident that for any target request, at any reasonable configuration of Autograph, our attack will succeed in a small number of rounds, even if content blocks from the target requests (and some related requests) are all blacklisted.

Finally, observe that the effectiveness of our attack drops when a (the average block size) increases. This is because with a larger a, the target request will be matched by fewer (but longer) content blocks in the blacklist. As a result, it is less likely for any random subsequence from the request to cross the boundary of two adjacent blacklisted blocks and result in a successful signature generation. On the other hand, the effectiveness of our attack increases with m (the minimum block size), and this trend is more significant for larger SEQ_LENs. This may be due to the following behavior of COPP: a separate content block with the last m bytes of the flow will be created if the normal partitioning does not find a content block that ends at the last byte of the

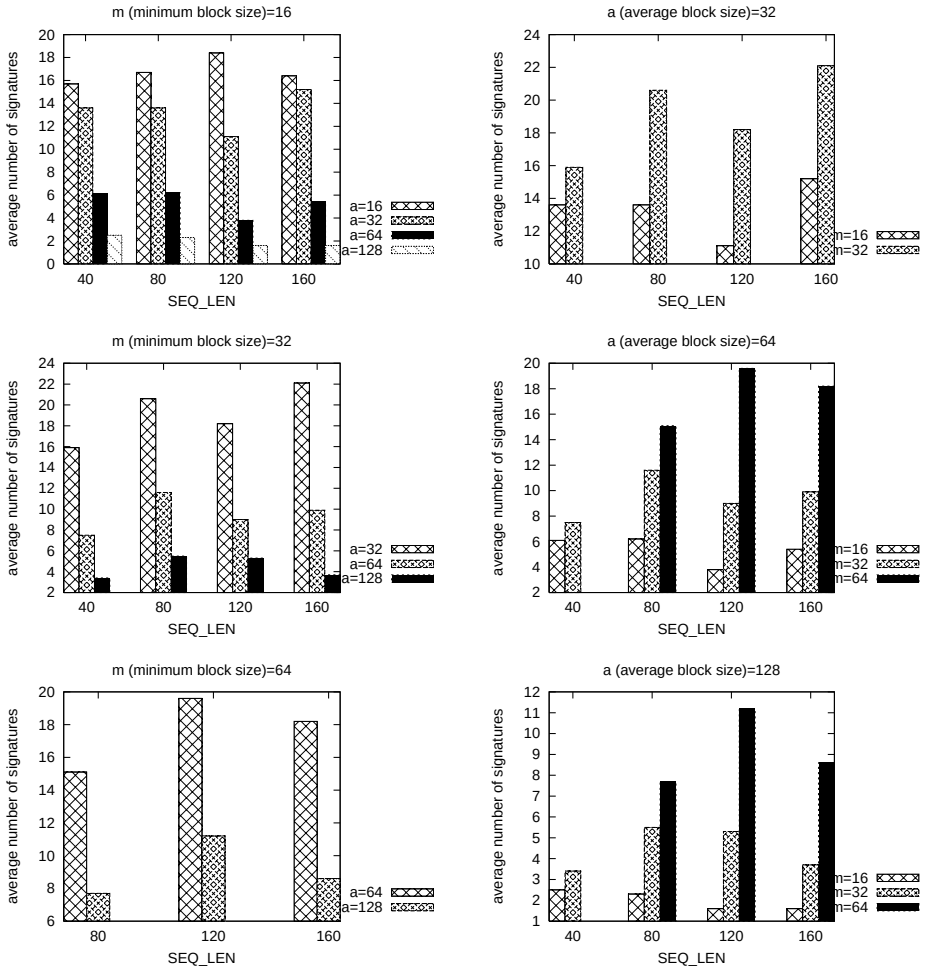

Figure B.1: The above figures show how the value of configuration parameters a and m of Autograph affect the effectiveness of our allergy attack at different SEQ_LEN. The effectiveness of our attacks is measured by the average number of distinct signatures generated in each of our 10 rounds of experiments. The column on the left shows the result of varying a while holding m constant, where the column on the right shows the effect of varying m while holding a constant. Note that no signatures will be generated when SEQ_LEN is smaller than m. Also note that we have only experimented on Autograph configurations with $a > m$.

suspicious flow. We believe, the last block thus generated, as well as the first block for our random subsequence have the best chance of being a new, non-blacklisted content block that achieves our goal. This is because both of them don't start after a 2-byte subsequence that matches the breakmark B. Thus, a longer m will mean a longer last block, which in turn increases the chance for it to cross the boundary of two adjacent blacklisted blocks. Furthermore, longer SEQ_LEN will mean that the content of the first and the last block are significantly different, and improve the chance that they will produce two separate blocks that have not been blacklisted.

Appendix C

The Broken Link Probability

In this appendix, we will give a precise definition of the Broken Link Probability (BLP), which is the metric we used to quantify the power of some of the type II and type III allergy attacks against websites that are presented in Chapter 2.

First of all, we'll define BLP as the probability that a user will click on a link to any unreachable page before the end of the user session (under the localized random surfer model presented in Sect. 2.3.1, this will mean the time between he/she reaches the root page of the site to the time when he/she gets bored and leaves). The BLP is intended to measure the degree of frustration (or inconvenience) caused by an allergy attack.

Before we present how we compute the BLP created by an allergy attack, we'll define a metric to measure the importance of a page under the localized random surfer model; we call this metric the "localized page rank". The computation of the localized page rank is the same as in [68], except that we do not normalize the page rank, and we initialize the page rank of the root to 1. We do not perform normalization because we are more interested in the actual number of times that a page will be visited, instead of its relative

importance among all other pages. The initial page rank of the root represents the visit to the root page that occurs at the beginning of each user session.

Now let us consider how we calculate the BLP. We start by recomputing the localized page rank for the website under attack. However, during this computation, pages made unavailable by the attack have a localized page rank of zero, though they are still counted as "children" of pages that link to them (without knowing which pages are blocked by an attack, visitors will behave as if there's no attack, and have equal chance of clicking on any link, broken or not). With the new set of localized page ranks, the BLP can be obtained by the following formula:

$$BLP = \sum_{p_i \in UR} d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}. \tag{C.1}$$

where UR is the set of pages made unreachable by the attack, $M(p_i)$ is the set of pages that have links to page p_i, $PR(p_i)$ is the localized page rank of the page p_i, and $L(p_i)$ is the number of pages pointed to by p_i. From the above formula, we see that the BLP is effectively the sum of page rank that the blocked pages inherit from pages that remain available under the attack. Note that while the localized page rank of a page is an overcount for the probability of visiting that page if it links to other pages to form a loop, it is not a problem for the BLP computation. This is because the user session ends on the first attempt to visit an unavailable page; i.e. an unreachable page can only be reached at most once in a user session. This also means visits

to various unreachable pages in a user session are mutually exclusive. Thus, we can compute the BLP by simply adding up the localized page rank of the unreachable pages.

Nomenclature

ASLR Address Space Layout Randomization, page 52

BLP Broken Link Probability, page 22

CIP Collaborative Intrusion Prevention, page 5

COPP Content-based Payload Partitioning, page 165

DEP Data Execution Prevention, page 85

DLL Dynamic Link Library, page 4

DMZ Demilitarized Zone, page 164

DoS Denial of Service, page 2

IDS(s) Intrusion Detection System(s), page 6

IPS(s) Intrusion Prevention System(s), page vi

ISR Instruction Set Randomization, page 48

LAIDS Lazy-able Intrusion Detection Systems, page 50

LIDS Lazy Intrusion Detection Systems, page 64

PC Program Counter, page 91

STEM Selective Transactional Emulation, page 57

VAD Virtual Address Descriptor, page 71

Bibliography

[1] The metasploit project.
http://www.metasploit.com/.

[2] Phoenix: Microsoft connect.
http://connect.microsoft.com/Phoenix.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 05)*, Virginia, Nov 2005.

[4] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS2006)*, Taipei, Mar 2006.

[5] J. Bethencourt, J. Franklin, and M. Vernon. Mapping internet sensors with probe response attacks. In *Proceedings of The 13th USENIX Security Symposium*, Aug 2005.

[6] B. Blakley. The emperor's old armor. In *Proceedings of the 1996 workshop on New security paradigms*, California, Sep 1996.

[7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings*

of The 2006 IEEE Symposium on Security and Privacy, Oakland, May 2006.

[8] F. Buchholz, T. Daniels, J. Early, R. Gopalakrishna, R. Gorman, B. Kuperman, S. Nystrom, A. Schroll, and A. Smith. Digging for worms, fishing for answers. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC 2002)*, Las Vegas, December 2002.

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, November 2006.

[10] S. Cho and S. Han. Two sophisticated techniques to improve hmm-based intrusion detection systems. In *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID2003)*, Pittsburgh, September 2003.

[11] S. P. Chung and A. K. Mok. On random-inspection-based intrusion dectection. In *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, Seattle, Sept 2005.

[12] S. P. Chung and A. K. Mok. Allergy attack against automatic signature generation. In *Proceedings of Ninth International Symposium on Recent Advances in Intrusion Detection (RAID2006)*, Hamburg, Sept 2006.

[13] S. P. Chung and A. K. Mok. Advanced allergy attacks: Does a corpus really help? In *Proceedings of Tenth International Symposium on Recent Advances in Intrusion Detection (RAID2007)*, Gold Coast, Sept 2007.

[14] S. P. Chung and A. K. Mok. Swarm attacks against network-level emulation/analysis. In *Proceedings of the Eleventh International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, Boston, September 2008.

[15] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, Brighton, Oct 2005.

[16] S. Coull, J. Branch, B. K. Szymanski, and E. Breimer. Intrusion detection: A bioinformatics approach. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, Lav Vegas, December 2003.

[17] C. Cowan and C. Pu. Death, taxes, and imperfect software: Surviving the inevitable. In *Proceedings of the 1998 Workshop on New Security Paradigms*, Virginia, Sep 1998.

[18] J. Crandall, Z. Su, S. Wu, and F. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 05)*, Virginia, Nov 2005.

[19] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Proceedings of GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) 2005*, Vienna, July 2005.

[20] D. E. Denning. An intrusion detection model. In *IEEE Transactions on Software Engineering*, Feb 1987.

[21] A. Edwards, H. Vo, A. Srivastava, and A. Srivastava. Vulcan: Binary transformation in a distribute environment. Number MSR-TR-2001-50, April 2001.

[22] eEye Digital Security. Sapphire worm code disassembled. http://www.eeye.com/html/Research/Flash/sapphire.txt, 2003.

[23] H. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Oakland, May 2003.

[24] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, May 1996.

[25] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, Seattle, Sept 2005.

[26] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using hidden markov models. In *Proceedings of Ninth International Symposium on Recent Advances in Intrusion Detection (RAID2006)*, Hamburg, Sept 2006.

[27] gera and riq. Advances in format string exploitation, July 2002.

[28] A. K. Ghosh, C. Michael, and M. Schatz. A real-time intrusion detection system based on learning program behavior. In *Proceedings of the Third International Symposium on Recent Advances in Intrusion Detection (RAID2000)*, October 2000.

[29] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, Sept 2005.

[30] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of 11th USENIX Security Symposium*, California, Aug 2002.

[31] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the Eleventh Network and Distributed System Security Symposium (NDSS2004)*, San Diego, February 2004.

[32] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In *Proceedings of the Ninth International Symposium on Recent*

Advances in Intrusion Detection (RAID2006), Hamburg, September 2006.

[33] Hitwise. http://www.hitwise.com.

[34] S. A. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. In *Journal of Computer Security, Vol. 6*, 1998.

[35] R. Hu and A. K. Mok. Detecting unknown massive mailing viruses using proactive methods. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection (RAID2004)*, France, Sept 2004.

[36] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the Third USENIX Windows NT Symposium*, Seattle, July 1999.

[37] Intel. Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide.
http://www.intel.com/Assets/PDF/manual/253669.pdf.

[38] B. P. Miller J. K. Hollingsworth and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High Performance Computing Conference (SHPCC)*, May 1994.

[39] Ken Johnson. Frame pointer omission (fpo) optimization and consequences when debugging.
http://www.nynaeve.net/?p=91.

[40] A. Jones and S. Li. Temporal signatures of intrusion detection. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, New Orleans, December 2001.

[41] Mark Wilton Jones. Browser speed comparison. http://www.howtocreate.co.uk/browserSpeed.html.

[42] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium*, California, August 2004.

[43] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of 11th USENIX Security Symposium*, California, Aug 2002.

[44] C. Ko. Logic induction of valid behavior specifications for intrusion detection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Berkely, May 2000.

[45] C. Kreibich and J. Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II)*, Boston, November 2003.

[46] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th Usenix Security Symposium*, Baltimore, August 2005.

[47] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Bayesian event classification for intrusion detection. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, Lav Vegas, December 2003.

[48] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the Eighth European Symposium on Research in Computer Security (ESORICS2003)*, Gjovik, October 2003.

[49] C. Krugel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, Sept 2005.

[50] L. Lam and T. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection (RAID2004)*, France, Sept 2004.

[51] T. Lane and C. Brodley. Temporal sequence learning and data reduction for anomaly detection. In *ACM Transactions on Information and System Security*, 1999.

[52] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of 7th USENIX Security Symposium*, San Antonio, January 1998.

[53] Z. Li, M. Sanghi, Y. Chen, M. Kao, and B. Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of The 2006 IEEE Symposium on Security and Privacy*, Oakland, May 2006.

[54] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 05)*, Virginia, Nov 2005.

[55] M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From stem to sead: Speculative execution for automated defense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, April 2007.

[56] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. Flips: Hybrid adaptive intrusion prevention. In *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, Sept 2005.

[57] Microsoft. Data execution prevention: frequently asked questions. http://windowshelp.microsoft.com/Windows/en-US/help/186de3d0-01af-4d4c-981d-674637d2f4bf1033.mspx.

[58] R. C. Miller and K. Bharat. SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers. In *Proceedings of 7th World Wide Web Conference*, Brisbane, April 1998.

[59] B. Moore, T. Slabach, and L. Schaelicke. Profiling interrupt handler performance through kernel instrumentation. In *Proceedings of the 2003 IEEE International Conference on Computer Design (ICCD03)*, San Joes, October 2003.

[60] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. http://www.caida.org/publications/papers/2003/sapphire/sapphire.html.

[61] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, San Francisco, Apr 2003.

[62] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *Proceedings of Tenth International Symposium on Recent Advances in Intrusion Detection (RAID2007)*, Gold Coast, Sept 2007.

[63] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of 13th Annual Network and Distributed System Security Symposium (NDSS 06)*, San Diego, Feb 2006.

[64] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of The 2005 IEEE Symposium on Security and Privacy*, Oakland, May 2005.

[65] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of Ninth International Symposium on Recent Advances in Intrusion Detection (RAID2006)*, Hamburg, Sept 2006.

[66] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb 2005.

[67] oldnewthing. In windows xp, even when dep is on, it's still sometimes off.
http://blogs.msdn.com/oldnewthing/archive/2007/11/16/6281925.aspx.

[68] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[69] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS2008)*, Tokyo, March 2008.

[70] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading Worm Signature Generators Using Deliberate Noise Injection. In *Proceedings of The 2006 IEEE Symposium on Security and Privacy*, Oakland, May 2006.

[71] M. Rajab, F. Monrose, and A. Terzis. Fast and evasive attacks: Highlighting the challenges ahead. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID2006)*, Hamburg, Sept 2006.

[72] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, May 2001.

[73] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 02)*, Washington, DC, Nov 2002.

[74] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Technical report, 2006.

[75] M. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection. In *Proceedings of Tenth International Symposium on Recent Advances in Intrusion Detection (RAID2007)*, Gold Coast, Sept 2007.

[76] S. Sidiroglou and A. Keromytis. Countering network worms through automatic patch generation, May 2005.

[77] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *Proceedings of the 2005 the USENIX Annual Technical Conference*, Apr 2005.

[78] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[79] S. Sidiroglou, M. Locasto, and A. Keromytis. Software self-healing using collaborative application. In *Proceedings of 13th Annual Network and Distributed System Security Symposium (NDSS 06)*, San Diego, Feb 2006.

[80] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, California, December 2004.

[81] skape. Understanding windows shellcode. http://www.hick.org/code/skape/papers/win32-shellcode.pdf, 2003.

[82] Sufatrio and R. H. C. Yap. "improving host-based ids with argument abstraction to prevent mimicry attacks. In *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, Sept 2005.

[83] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In

Proceedings of Fifth International Symposium on Recent Advances in Intrusion Detection (RAID2002), Zurich, October 2002.

[84] K. M. C. Tan and R. A. Maxion. "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, May 2002.

[85] B. Tancer. Obama clinton chart updated with edwards. http://www.hitwise.com/datacenter/industrysearchterms/all-categories.php, Jan 2007.

[86] E. Totel, F. Majorczyk, and L. Me. Cots diversity intrusion detection and application to web servers. In *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, Sept 2005.

[87] P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In *Proceedings of the Third International Symposium on Recent Advances in Intrusion Detection (RAID2000)*, October 2000.

[88] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, May 2001.

[89] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 02)*, Washington, DC, Nov 2002.

[90] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM 2004*, Portland, Aug 2004.

[91] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, Sept 2005.

[92] Y Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of 13th Annual Network and Distributed System Security Symposium (NDSS 06)*, Feb 2006.

[93] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, May 1999.

[94] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the Third International Symposium on Recent Advances in Intrusion Detection (RAID2000)*, October 2000.

[95] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the 18th Annual Computer*

Security Applications Conference (ACSAC 2002), Las Vegas, December 2002.

[96] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection (RAID2004)*, France, Sept 2004.

[97] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 05)*, Virginia, Nov 2005.

[98] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of 14th USENIX Security Symposium*, Maryland, Aug 2005.

Vita

Simon Pak Ho CHUNG received his Bachelor of Engineering degree from the University of Hong Kong in 2001. He then moved to the US for his graduate studies in the University of Texas at Austin in September, 2001. He obtained his Master of Science degree in Fall 2003, and is expected to obtain his PhD degree in August 2009. His main research interest is in intrusion detection/prevention systems.

Permanent address: 2518, Leon Stree, Apt 206
Austin, Texas 78705

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.